UNIVERSITÀ DEGLI STUDI DI PADOVA

Facoltà di Scienze MM.FF.NN

Corso di Laurea Triennale in Informatica

Anno Accademico 2008-2009

Integrazione di Google Document in Applicazioni Web

Relatore: Prof. Gilberto Filè

Laureando: Davide Compagnin

Indice

Convenzioni tipografiche

Questo documento è stato realizzato tramite lo strumento Microsoft Office Word 2007 e basato su un font eco-compatibile (www.ecofont.eu/ecofont_en.html). Con lo scopo di facilitare la lettura saranno utilizzate le seguenti convenzioni tipografiche:

- *Carattere corsivo:* per indicare termini tecnici, di lingua inglese e non tradotti.
- `Carattere costante:` per indicare linee di codice Java o Javascript.

1 Introduzione

1.1 Scopo del documento

Questo documento costituisce la relazione sulle attività svolte durante il periodo di stage presso l'azienda Zucchetti S.p.A., nonché la tesi finale per il corso di laurea triennale in Informatica. L'azienda padovana presso cui ho operato si occupa principalmente dello sviluppo di applicazioni gestionali personalizzate e per ciò le esigenze dell'azienda si possono ricondurre a necessità di gestione e pubblicazione delle informazioni, che richiedono un forte impegno nella ricerca di nuove soluzioni in grado di semplificare ed automatizzare queste operazioni. In particolare, la pubblicazione rappresenta un ambito operativo molto importante all'interno di un sistema informativo poiché permette la distribuzione e la condivisione rapida delle informazioni. Spesso l'operazione di pubblicazione è preceduta da un processo molto complesso di analisi, elaborazione dati e informazioni atomiche in un insieme di conoscenze ordinate, coerenti e riassuntive. In un contesto prettamente gestionale è però spesso indispensabile avere strumenti adeguati che permettano di esportare anche frequentemente i risultati dei processi aziendali tramite la creazione di documenti. Per questo, il lavoro da me svolto aveva come obiettivo quello di verificare l'integrabilità del servizio *Google Document* attraverso lo sviluppo di un sistema per la pubblicazione di informazioni da un applicativo Web. Più precisamente, il compito da me svolto è stato quello di sviluppare una componente che, inserita all'interno di una generica applicazione Web, è in grado di interagire con il servizio remoto di Google per la creazione e la gestione di documenti. In particolare, la componente sviluppata costituisce un'interfaccia di comunicazione che permette la trasformazione di un flusso di dati uscente da un'applicazione verso un documento remoto. Oltre a ciò, questa componente è stata successivamente utilizzata nella creazione di una libreria per il software *SitePainter*, che permette di automatizzare le operazioni di integrazione durante lo sviluppo delle applicazioni. Bisogna sottolineare che questa componente può essere aggiunta o meno in base alle esigenze dell'azienda. Una parte del progetto, di aspetto più teorico, è stata inoltre dedicata all'analisi delle problematiche di sicurezza quando si opera con le applicazioni

Web. Nel dettaglio, si sono analizzati gli aspetti problematici che riguardano l'autenticazione degli utenti per l'accesso all'account Google. Si è voluto implementare un'infrastruttura minimale in grado di operare con un protocollo specifico per le applicazioni Web chiamato *AuthSub*, al fine di rendere completamente operativa la componente. Riassumendo, si è voluto creare una componente in grado di integrarsi in un'applicazione che permetta inoltre di pubblicare dei dati sottoforma di documenti presso un account Google. Lo stage ha visto la suddivisione del lavoro in varie fasi. Dapprima è stato necessario compiere un'approfondita analisi dell'ambiente operativo e delle tecnologie utilizzate. In seguito si è analizzato l'apprendimento del servizio *Google Document*, delle sue librerie ed infine si è proceduto allo sviluppo della componente e del sistema di autenticazione. Il documento redatto, che descrive i passi fondamentali del lavoro svolto, è stato perciò suddiviso in due parti principali: una prima parte definisce gli obiettivi generali, ciò che si vuole ottenere, e analizza i requisiti; mentre una seconda parte definisce le attività svolte per il raggiungimento degli obbiettivi. Ognuna di queste parti a sua volta è suddivisa in due fasi. La Fase I riguarda lo sviluppo della componente e alcuni esempi d'uso, mentre la Fase II illustra le principali problematiche di sicurezza delle applicazioni Web e analizza in dettaglio il protocollo di autenticazione *AuthSub*. Saranno inoltre evidenziate le problematiche emerse durante i lavoro ed infine le scelte effettuate per raggiungere gli obbiettivi preposti. Per quanto riguarda gli aspetti informatici fondamentali, questo tipo di lavoro ha richiesto la conoscenza del linguaggio di programmazione Java, del protocollo HTTP, del linguaggio XML e l'applicazione di queste conoscenze in merito allo sviluppo della componente. Del progetto, il cui committente è la stessa azienda Zucchetti S.p.A., me ne sono occupato direttamente ed esclusivamente, svolgendo le attività di creazione, analisi, progettazione e sviluppo della componente Java.

1.2 Dominio applicativo

In questa parte è descritto il dominio applicativo, cioè il contesto in cui opera l'azienda e l'insieme degli strumenti con i quali ho operato.

1.2.1 L'azienda

Zucchetti S.p.A. rappresenta uno dei maggiori protagonisti dell'IT sul panorama italiano. Essa si occupa principalmente di soluzioni

software, hardware e servizi innovativi realizzati e studiati per soddisfare le specifiche esigenze di aziende di qualsiasi settore e dimensione, banche e assicurazioni, professionisti, associazioni di categoria, pubblica amministrazione, ecc. L'azienda è dislocata in vari dipartimenti ma lo stage è stato tenuto nella sede di Padova dove si svolgono prevalentemente attività di ricerca ed innovazione sui prodotti. Questa sede conta attualmente di una decina di elementi specializzati in ambiti diversi ma che si occupano principalmente alla manutenzione di software per lo sviluppo automatizzato di applicazioni complesse.

1.2.2 Software per lo sviluppo automatizzato

Per rendere competitivo un prodotto software è indispensabile avere degli strumenti che consentano di ridurre i tempi di realizzazione e manutenzione mantenendo però elevato il livello qualitativo. Effettuare una costante ricerca e innovazione e disporre continuamente di nuove tecnologie è la via migliore per raggiungere questi obbiettivi. L'azienda attualmente dedica molti sforzi nel migliorare le tecniche e gli strumenti per automatizzare la creazione di programmi gestionali sempre più complessi facendo in modo che l'utilizzatore concentri l'attenzione prevalentemente sull'analisi delle funzionalità e sulla progettazione ad alto livello astraendo il più possibile la progettazione di dettaglio e la codifica.
Un'applicazione generata tramite uno strumento di sviluppo di questo tipo tipicamente è riconducibile ad uno schema di tipo MVC. Questo pattern architetturale, molto diffuso prevede tre componenti:

- Modello: Rappresenta il nucleo centrale dell'applicazione, contiene lo schema secondo cui i dati sono in relazione tra loro e tutti i metodi per accedervi e manipolarli. Questa componente è logicamente indipendente dalle altre due e di fatto è implementata tramite un DBMS sempre attivo durante l'esecuzione del programma.
- Vista: In base alle richieste dell'utente presenta come risultato una vista sui dati. L'utilizzatore solitamente possiede un accesso diretto a questa componente tramite un'interfaccia grafica tradizionale oppure una visualizzazione basata su pagine Web dinamiche supportata dalla nascita di nuovi strumenti, come ad esempio AJAX.
- Controllo: Modifica lo stato delle altre due componenti coordinandone le attività, raggruppa tutte le funzionalità

necessarie per l'interpretazione corretta delle richieste da parte dell'interfaccia e organizza in maniera adeguata tutte le modifiche imposte al modello.

Questo schema, tra l'altro, implica una separazione fra la logica applicativa, che in questo contesto spesso è chiamata logica di business, a carico del controllo e del modello, e l'interfaccia utente a carico della vista. Quindi ricopre perfettamente quella che è la struttura di una tipica applicazione Web, in cui vi è l'unione eterogenea di più parti e tecnologie che devono interagire tra loro. La Figura 1 visualizza proprio questo schema.

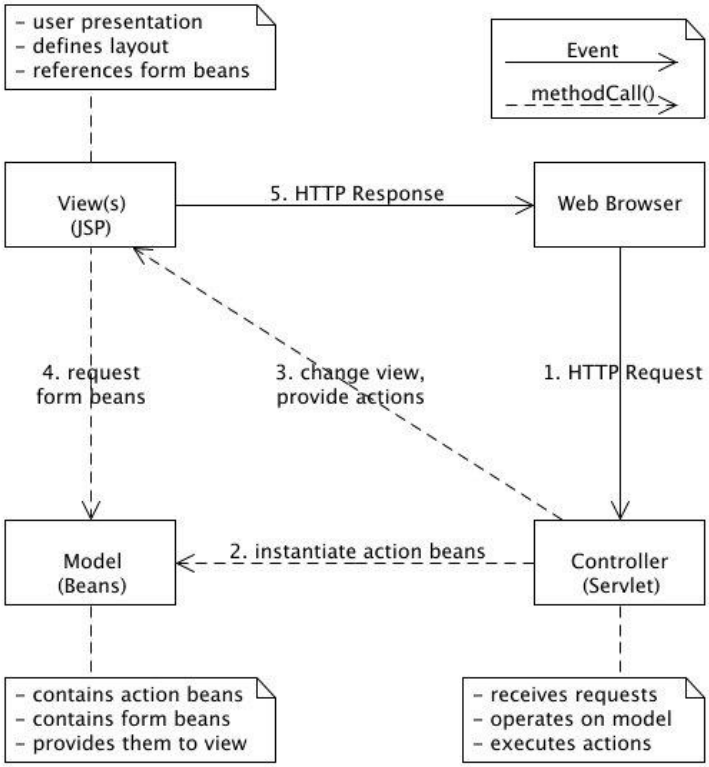


Figura 1

La realizzazione di applicazioni gestionali seppure semplici costituisce un'operazione onerosa in quanto tipicamente le parti che si devono costituire sono molte, tuttavia queste seguono un modello applicativo ben definito e quindi è possibile utilizzare degli strumenti che ne automatizzino la generazione. A titolo d'esempio possiamo citare la definizione e il mantenimento della base di dati, piuttosto che il controllo dell'interfaccia grafica oppure la computazione dei dati, tutte operazioni che molto spesso necessitano l'uso di linguaggi di programmazione e tecnologie diverse. In questi casi è essenziale l'utilizzo di formalismi grafici e di strumenti visuali che rendano sempre più intuitivi e sistematici concetti complessi, piuttosto di una produzione manuale del codice sorgente. Attualmente l'azienda è

concentrata nello sviluppo dei seguenti applicativi, oggetto peraltro di studio in questo progetto:

- *PortalStudio*
- *SitePainter*

Questi prodotti in realtà non sono indipendenti ma interagiscono tra loro con l'obbiettivo di ridurre drasticamente tempi e costi di produzione e nello stesso tempo di governare la complessità delle nuove tecnologie. *SitePainter* è un sistema di sviluppo specializzato nella realizzazione di applicazioni transazionali fortemente interattive in ambiente Web (*n-tier*) con interfaccia browser. È scritto in C++ ed esegue localmente su un computer, mentre *PortalStudio* è un ambiente di sviluppo *Web-RAD* (*Web Rapid Application Development*) per la realizzazione di complesse applicazioni per la pubblicazione e divulgazione di dati (portali aziendali, siti Web, ecc.) in ambiente Web con interfaccia browser. Quest'ultimo si presenta come modulo aggiuntivo di *SitePainter* che una volta importato può essere generato. Questa operazione corrisponde ad una sorte di compilazione avanzata e consiste sostanzialmente nell'effettuare la creazione di tutte le *servlet* Java, le pagine JSP, il codice Javascript, le pagine HTML, gli stili CSS e infine il *deploy* nel *Web-server*. È importante sottolineare come questa procedura sia del tutto generale per ogni sistema basato su tecnologia *Java Enterprise*, e quindi l'ho utilizzata molte volte nel corso del progetto quando ho sviluppato prototipi di applicazioni Web. Analizzo ora il funzionamento nel dettaglio di questi due strumenti.

1.2.3 SitePainter

Appartiene alla categoria di software *i-C.A.S.E.* (*Internet Computer Aided Software Engineering*). Offre un insieme di strumenti visuali perfettamente integrati per gestire e coordinare tutte le fasi del processo di sviluppo e manutenzione del software: l'analisi, il disegno della base dati, l'interfaccia utente, la logica applicativa, la manutenzione correttiva fino alla stesura della manualistica utente e tecnica.

Può gestire più progetti contemporaneamente e indipendentemente, organizzati in directory.

Queste sono alcune delle funzionalità offerte da *SitePainter* che sono state utilizzate in questo stage:

- Strumento grafico per il disegno della struttura della base di dati tramite la creazione di tabelle, la definizione dei vincoli relazionali e l'impostazione dei permessi utente.
- Strumento per la generazione automatica di un portale Web dinamico secondo la struttura della base di dati e contenente strumenti avanzati per:
    - Il controllo e manutenzione della base dati in corso d'opera.
    - Il controllo degli errori e delle prestazioni del sistema.
    - L'immissione e la cancellazione di dati e di tabelle sul database tramite interfacce e maschere personalizzabili.
    - La realizzazione di interrogazioni anche molto complesse.
- Strumento per la creazione di maschere e personalizzazione di interfacce per l'inserimento e la visualizzazione per il portale Web.
- Strumento di *Routine Painter* per la creazione di routine.
- Strumento di *Library Editor* per la creazione e l'importazione di librerie esterne.

La struttura di base di ogni applicazione sviluppata tramite *SitePainter* è costituita da un portale per la gestione della base di dati, di fatto un'applicazione *Web*. Una delle funzionalità sempre presente, in forma più o meno sofisticata, è rappresentata dal sistema di interrogazione e visualizzazione dinamica dei dati che prende il nome di *Zoom*. Sostanzialmente è una griglia che tramite un pannello permette di settare un gran numero di opzioni e di definire dei parametri secondo i quali filtrare le informazioni estratte dalla base di dati.

Di fondamentale importanza sono stati gli strumenti di *Routine Painter* e *Library Editor*. Il primo permette di definire, come dice il nome stesso, delle routine cioè delle procedure o funzioni di elaborazione (*batch*) che interagiscono da un lato con la base di dati, il Modello, e dall'altro con il resto dell'applicazione, il Controllo rispettivamente al pattern MVC. Attraverso questo strumento diventa estremamente facile quindi implementare operazioni non previste dalla logica applicativa predefinita, e grazie al legame con la base di dati è possibile esprimere qualsiasi funzionalità richiesta dall'applicazione. Il principio logico è tanto semplice quanto potente ed è mostrato in Figura 2. La definizione di una routine corrisponde

alla definizione di una *query*, in un linguaggio simile ad SQL, in cui i dati in ingresso oppure quelli prodotti possono essere assegnati a variabili logiche e quindi fungono da parametri per la *query* stessa ed inoltre possono essere importati o esportati dalla routine per essere utilizzati in altri contesti del programma.

Figura 2

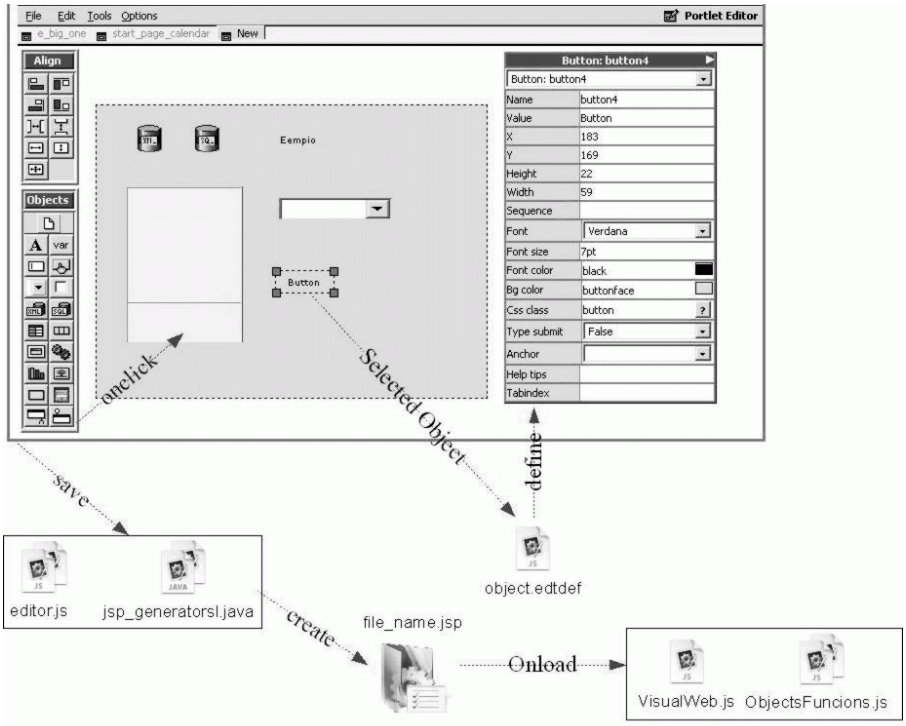
Come una routine viene generata e inserita nell'applicazione è spiegato poi nel capitolo 2.1.1.

Il secondo strumento di fondamentale importanza è il *Library Editor*, in quanto consente di definire una libreria le cui funzionalità possono essere richiamate dall'interno degli altri strumenti di *SitePainter*, come ad esempio dal disegnatore di routine. Questo mi ha permesso di integrare la componente Java sviluppata, direttamente nell'ambiente di sviluppo. Il funzionamento di questo strumento è molto semplice e consiste in una serie di finestre tramite le quali è possibile definire dei metodi come fossero oggetti, specificandone il nome, il tipo dei parametri, il tipo del valore ritornato e il codice che ne compone il corpo. Dopodiché viene generato uno scheletro che raggruppa tutti i metodi, ne salva la configurazione tramite un file XML e il codice sorgente Java.

Vedremo nel capitolo 3.1 come l'utilizzo di questi due strumenti permettano di sviluppare una applicazione Web, che utilizzi le funzionalità fornite dalla senza produrre alcuna linea di codice manualmente. Questo ha rappresentato uno degli obbiettivi principali portati a termine.

1.2.4 PortalStudio

PortalStudio è un applicativo completamente orientato al *Web* e quindi è utilizzabile soltanto tramite *Web-browser*.



L'immagine rappresenta uno schema di funzionamento di *PortalStudio*. Una pagina dinamica sviluppata con questo strumento prende il nome di *Pagelet*, ed è una composizione di più moduli chiamati *Portlet*. Ognuno di questi definisce una funzionalità e può comunicare con gli altri moduli presenti nella stessa pagina grazie ad un meccanismo di eventi. Una volta creato un *Portlet* è possibile aggiungere, eliminare e modificare i singoli oggetti che lo compongono. Ognuno di questi ha un comportamento e delle funzionalità differenti specificate in file Javascript collegati all'applicazione, quindi nel caso questo necessiti di un comportamento non di base è possibile modificarlo manualmente specificandone il codice tramite una finestra apposita. Ogni oggetto può essere istanziato più volte e può dipendere da altri oggetti secondo le proprietà definite sull'oggetto stesso. Gli oggetti che compongono un *Portlet* possono essere sia di interfaccia, come bottoni, caselle di testo, griglie ecc. e sono visibili nella pagina generata, oppure funzionali come variabili,
XMLDataObject, *SQLDataObject*, *SPLinker*, ecc.. L'interfaccia di *PortalStudio* possiede un pannello *Object* dal quale è possibile aggiungere oggetti e un pannello *Properties* che ne permette la modifica delle proprietà. Quando un utente salva il contenuto di una *Portlet* viene generato un file JSP nel quale vengono memorizzati tutti gli oggetti e le relative proprietà. Le componenti più frequentemente utilizzate sono state:

- *XMLDataObject*: Questa componente permette un collegamento tra una sorgente XML e le componenti di visualizzazione. Ad esempio è possibile richiamare una *servlet* che genera un documento XML interpretato poi *dall'XMLDataObject* secondo la definizione di regole *XPath* può essere inserito in una griglia piuttosto che una casella di testo ecc.
- *SPLinker*: Questa componente ha molteplici usi, è completamente definibile e permette un collegamento tra la logica dell'applicazione lato server e gli oggetti di una *Portlet*. Nel nostro caso ad esempio è stata utilizzata per richiamare una routine passando alcuni parametri definiti come variabili secondo quanto descritto nel capitolo 1.2.3.
- Variabili: Come in un linguaggio di programmazione permettono la memorizzazione di informazioni temporanee che dipendono da una specifica esecuzione dell'applicazione. Queste non saranno visibili dal *Web-browser* nella pagina prodotta.

1.3 Tecnologie e strumenti utilizzati

Per lo sviluppo del progetto è stata messa a disposizione una postazione dedicata nella quale sono stati installati localmente gli applicativi aziendali sui quali compiere le varie prove, senza causare conflitti al server centralizzato. Il sistema operativo di riferimento è stato "Microsoft Windows XP".

1.3.1 Javascript

Javascript è il linguaggio di *scripting* orientato agli oggetti ad oggi più diffuso e supportato nei siti Web. Questo linguaggio mi ha permesso di creare pagine interattive e implementare la logica dell'applicazione nel lato client. Nel nostro caso viene generato dal server in maniera dinamica per gestire gli eventi legati agli oggetti della pagina, e viene interpretato dal Browser per visualizzare i risultati. Una caratteristica importante di questo linguaggio è che è debolmente tipizzato quindi in generale non è possibile definire il tipo di una variabile in fase di dichiarazione della variabile stessa, ma esso viene inferito direttamente dall'interprete in maniera automatica.

1.3.2 Servlet

Una *servlet* è un oggetto Java, che deve rispettare determinate specifiche, che processa una richiesta HTTP rispondendo con dell'output, tipicamente una pagina Web. Una *servlet* in generale agisce nella parte Controller del pattern MVC e si occupa direttamente di processare le richieste cioè di implementare la logica del programma dal lato del server. Nel nostro caso sono state utilizzate in molte situazioni sia per gestire l'autenticazione tramite il protocollo *AuthSub* sia come *proxy* tra il provider Google e l'applicazione client. Esegue tipicamente su un Web container.

1.3.3 Apache Tomcat

È un Web container open source sviluppato dalla Apache Software Foundation. Implementa le specifiche JSP e *servlet* di Sun, fornendo quindi una piattaforma per l'esecuzione di applicazioni Web sviluppate nel linguaggio Java. Rappresenta il sistema di riferimento utilizzato dall'azienda tanto che è stato integrato direttamente nello strumento *SitePainter* dal quale è possibile comandarne l'avvio, l'arresto e persino automatizzare la fase di *deploy* dell'applicazione.

1.3.4 NetBeans

Per la realizzazione del progetto ho scelto NetBeans come ambiente di sviluppo. Un motivo è perché possedevo già una buona familiarità con questo strumento, mentre un altro è perché questo integra un ottimo plug-in di sviluppo con UML. NetBeans, è un ambiente di sviluppo creato da Sun, che possiede svariate funzionalità ed essendo ormai alla versione 6.1 può definirsi abbastanza consolidato.

2 Definizione del problema

2.1 Architettura delle applicazioni Web

La prima fondamentale operazione eseguita durante lo stage è stata quella di studiare il funzionamento e l'architettura delle applicazioni Web, principalmente tramite materiale online. Questo tipo di applicazioni si distingue da quelle tradizionali poiché è distribuita su una rete e accessibile da un *Web-browser*.

Tipicamente in ogni applicazione *Web* è possibile identificare una suddivisione logica e una architetturale. La prima è definita dal modello client-server in cui il server rappresenta il fornitore dell'applicazione mentre i client sono rappresentati dai vari *Web-browser* degli utenti. Mentre la seconda è determinata dall'utilizzo del pattern MVC come descritto nel capitolo 1.2.2. Quest'ultimo permette di isolare le varie componenti architetturali e di renderle indipendenti dall'uso di una tecnologia specifica.

La struttura di ogni applicazione definita secondo i vicoli definiti nel punto 1.2.2 è riconducibile allo schema evidenziato in Figura 1 e caratterizzata dalle seguenti associazioni:

- Modello: Definito dall'uso di *Beans* ossia delle classi che permettono di rappresentare le informazioni contenute all'interno di una base di dati. Non hanno riguardato direttamente lo svolgimento di questo progetto.
- Vista: Definita dall'insieme di pagine JSP, usate per la costruzione dinamica delle pagine HTML costituenti l'interfaccia utente.
- Controllo: Definito dalle *servlet* ovvero le componenti fondamentali per la gestione delle richieste degli utenti.

Ogni applicazione Web definita secondo questo modello utilizza solitamente tecnologie diverse per ognuna delle componenti. Negli esempi sviluppati all'interno di questo progetto è possibile identificare l'uso di *servlet* e JSP lato server, mentre una composizione di HTML, CSS, e Javascript lato client. Un programma quindi è una composizione di queste parti. Naturalmente queste interagiscono tra loro attraverso la rete e quindi necessitano del supporto adeguato al loro funzionamento, costituito dal *Web container* Apache Tomcat.

2.1.1 Generazione delle applicazioni

Dopo aver analizzato l'infrastruttura generale, è importante capire come gli strumenti *SitePainter* e *PortalStudio* generano una applicazione a partire dalla progettazione delle componenti, cioè capire cosa avviene quando si costruisce una *Portlet* piuttosto che una routine ecc. In generale una applicazione sviluppata con *SitePainter* di base rappresenta un portale Web, come descritto nel capitolo 1.2.3 costituito da una aggregazione di *Portlet*, dal quale si parte per espanderne le funzionalità. Ogni *Portlet* costituisce una pagina JSP, il cui codice è generato a partire dagli elementi contenuti al suo interno. Ogni oggetto quindi genera una sequenza di istruzioni Java e Javascript che vanno a comporre dinamicamente il comportamento della pagina quando eseguita nel browser. Allo stesso modo funzionano le routine, la cui generazione consiste nella creazione di una *servlet*. All'interno è definito il codice per eseguire le operazioni fondamentali come la lettura dei parametri, la connessione al database e l'esecuzione delle *query*. Tutti gli elementi sviluppati quindi convergono nell'uso delle medesime tecnologie e questo ne facilita la manutenzione.

2.2 Il servizio Google Document

2.2.1 Panoramica

Google Document è un servizio online per la produzione e gestione di documenti. È una raccolta di applicazioni che permettono di gestire principalmente fogli di calcolo, documenti di testo e presentazioni. La vera peculiarità è di essere una applicazione Web quindi di risiedere sul server Google e può essere lanciata da remoto non richiedendo l'installazione di software sul computer locale. Nemmeno i dati sono salvati in locale, consentendo di condividere il file con altri utenti a diversi livelli di privilegio (sola lettura, accesso in scrittura ad alcune parti o a tutto il documento) e di accedervi da qualunque computer collegato ad Internet. Una funzionalità molto importante è la collaborazione, tramite la quale i documenti possono essere condivisi, aperti e modificati da più utenti nello stesso istante.

2.2.2 Google Data API

Recentemente è disponibile un insieme di API che permette l'accesso ai servizi anche al di fuori del portale Web principale, quindi di scrivere applicazioni in grado di utilizzare le funzionalità disponibili.

Analizzare il funzionamento di queste librerie mi ha permesso di sviluppare la componente Java, obbiettivo di questo progetto. La libreria principale su cui si basa il servizio prende il nome di *GData*. Si tratta di un protocollo per la lettura e scrittura di dati sul Web, infatti associa il protocollo HTTP ad un formato di interscambio di dati basato su XML, in particolare *Atom* e RSS, per creare un'interfaccia che segue i principi dell'architettura REST, ovvero:

- Lo stato dell'applicazione e le funzionalità sono divisi in risorse.
- Ogni risorsa è unica e indirizzabile usando sintassi universale.
- Tutte le risorse condividono un' interfaccia uniforme per l'interazione e consiste di
    - un insieme vincolato di operazioni ben definite
    - un insieme vincolato di contenuti

Così è possibile utilizzare i metodi messi a disposizione dal protocollo HTTP per eseguire qualsiasi operazione, come ad esempio:

- GET:richiede una rappresentazione della risorsa specificata
- POST:permette di scrivere dei dati nella risorsa specificata
- PUT:crea la rappresentazione per la risorsa specificata
- DELETE:elimina la risorsa specificata

L'elemento centrale ad ogni modo è rappresentato dalle risorse, la cui rappresentazione rispetta il formato *Atom feed*. Un *feed* costituisce una porzione di informazione unitaria e strutturata. Intuitivamente ogni documento costituisce una risorsa e quindi anche un *feed*. Il vantaggio di utilizzare queste semplici tecnologie garantisce l'indipendenza dalla piattaforma e quindi permette l'utilizzo di un qualsiasi linguaggio di programmazione in grado di gestire il protocollo http, per questo ho avuto la possibilità di utilizzare il linguaggio Java visto che lo supporta appieno. Tuttavia gestire la comunicazione fin dai livelli più bassi può essere molto dispendioso in termini di tempo, quindi Google fornisce un'implementazione delle proprie librerie per svariati linguaggi di programmazione tra cui Java, le quali mettono a disposizione delle classi e delle strutture dati per facilitare l'uso del servizio nascondendo qualsiasi dettaglio riguardante il protocollo sottostante. La struttura generale è mostrata in Figura 3 ed essenzialmente comprende un livello più basso in qui vi è appunto la gestione generalizzata della comunicazione, sul quale si appoggia un livello specializzato ai vari servizi e alle varie funzionalità.

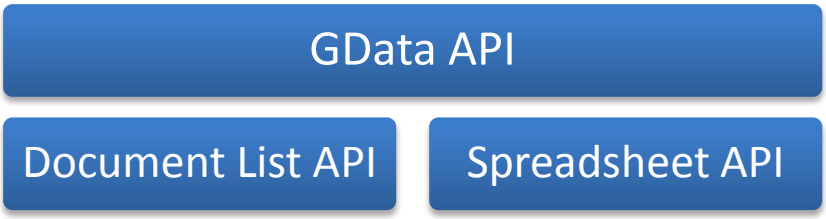


Figura 3

Durante l'analisi ho trovato queste librerie piuttosto intricate forse a causa della scarsa documentazione a disposizione, per qui molto tempo è stato impiegato all'apprendimento sull'uso, ma ai fini del progetto è stato sufficiente capirne il funzionamento soltanto di alcune parti. In particolare ho notato una struttura ripetitiva e una forte corrispondenza con i principi analizzati precedentemente. Essenzialmente vi sono tre classi principali: *Service*, *Feed* ed *Entry*. La prima fornisce i metodi per la comunicazione, mentre le altre due forniscono una rappresentazione ad oggetti delle risorse. Vediamo come la stessa operazione di inserimento di un nuovo documento vuoto possa essere effettuata in tre modi differenti. Il primo esempio mostra l'uso del protocollo HTTP grezzo, il secondo mostra la connessione al servizio senza l'utilizzo delle API Google che invece sono utilizzate nel terzo. Il secondo esempio è stato preso da una classe da me costruita durante le prime fasi del progetto per testare la fattibilità di un livello di connessione indipendente dalle librerie come descritto anche durante la Fase I.

```
POST http://docs.google.com/feeds/documents/private/full HTTP/1.1
Content-Length:      288
Content-Type: application/atom+xml
Authorization: AuthSub token="token di autorizzazione"
<?xml version='1.0' encoding='UTF-8'?>
 <atom:entry xmlns:atom="http://www.w3.org/2005/Atom">
  <atom:category scheme="http://schemas.google.com/g/2005#kind"
term="http://schemas.google.com/docs/2007#document"/>
  <atom:title>Documento1</atom:title>
 </atom:entry>
```

```
public static void crea_nuovo() {
URL url=new URL("http://docs.google.com/documents/private/full");
HttpURLConnection conn=(HttpURLConnection) url.openConnection();
conn.setRequestProperty("Content-Type","application/atom+xml");
conn.setRequestProperty("Authorization","AuthSub token=token");
OutputStreamWriter out = new
OutputStreamWriter(conn.getOutputStream());
out.write("
 <?xml version='1.0' encoding='UTF-8'?>
 <atom:entry xmlns:atom='http://www.w3.org/2005/Atom'>
  <atom:category scheme='http://schemas.google.com/g/2005#kind'
term='http://schemas.google.com/docs/2007#document'/>
  <atom:title>Documento1</atom:title>
 </atom:entry> ");
out.close();
}
```

```
public static void crea_nuovo() {
DocsService service=new DocsService();
DocumentEntry nuovo=new DocumentEntry();
nuovo.setTitle(“Documento1”);
URL url=new URL(“http://docs.google.com/documents/private/full”);
service.insert(url,nuovo);
}
```

È facile notare le analogie tra il primo e il secondo esempio. Leggermente diverso invece è il terzo, che nasconde l'utilizzo del protocollo HTTP in quanto gestito dalla classe *DocsService*. Inoltre la creazione del codice XML per ogni tipologia di documento è definita dalla corrispondente classe di Entry: *DocumentEntry*, *SpreadsheetEntry* e *PresentationEntry*.

Una precisazione necessaria riguarda la struttura di uno spreadsheet. In quanto è definito come una collezione di *worksheet* cioè fogli di lavoro costituiti a sua volta dalle celle. Per questo esistono altre due classi *WorksheetEntry* e *CellEntry* che permettono appunto questa distinzione.

2.2.3 Gestione delle risorse

Considerato il fatto che ogni documento costituisce una risorsa un punto molto importante rappresenta l'identificazione di queste all'interno del servizio. Questa operazione avviene tramite la composizione di due elementi, ovvero da un URL di base comune e da una chiave univoca. Ad esempio:

```
http://docs.google.com/feeds/documents/private/full/document/dd79pp22_52mjxsmc9
```

rappresenta l'ID, ma soltanto "dd79pp22_52mjxsmc9" costituisce la chiave. Questo è molto importante da tenere conto poiché svariati metodi della componente che ho sviluppato fanno riferimento proprio a questa chiave per identificare un documento. Diverso invece è il collegamento con il quale l'applicazione *Google Document* ne risolve l'apertura, e questo prende la forma seguente:

```
http://docs.google.com/Doc?id=dd79pp22_52mjxsmc9
```

Il primo indirizzo rappresenta la locazione precisa dove la risorsa è situata, mentre il secondo rappresenta la locazione dell'applicazione parametrizzata sul documento da mostrare.

2.3 Fase I

Questa fase ha riguardato lo sviluppo della componente, dalla definizione dei requisiti e dall'analisi delle possibili alternative architetturali all'implementazione vera e propria e l'inserimento in qualche contesto applicativo d'esempio. È stata la più onerosa dal punto di vista temporale ed stata caratterizzata da alcune scelte fondamentali che saranno trattate in questa sezione.

Gli obbiettivi di questa analisi rappresentano la creazione di una componente in grado di esportare le informazioni contenute in un'applicazione web all'interno di un account Google sottoforma di documento. In generale un'applicazione web gestisce un gran numero di informazioni provenienti sia dal sistema di immagazzinamento dei dati, ma anche statistiche sull'accesso, reportistiche d'errore ecc. e talvolta vi è l'esigenza di pubblicarle archiviarle o produrre copie cartacee. Per far ciò spesso ci si affida ad un DMS ovvero un sistema in grado di memorizzare, catalogare, condividere documenti tra più utenti. Ebbene anche il servizio *Google Document* appartiene a questa categoria in quanto fornisce funzionalità di archiviazione, indicizzazione, supporto ai metadati, recupero, versionamento, collaborazione, pubblicazione di documenti e inoltre integrazione con le applicazioni di *office automation* più diffuse il tutto contornato da un efficiente accesso via Web. La componente quindi deve essere in grado di mettere in comunicazione un'applicazione Web con questo servizio e fornire dei metodi in grado di automatizzare il processo di trasferimento di dati all'interno dei documenti. Inoltre la sua architettura deve essere strutturata a livelli

poiché è necessario gestire più in basso la comunicazione attraverso il protocollo HTTP mentre più in alto le funzionalità da offrire che ho classificabili in tre categorie:

- Gestione dell'accesso degli utenti tramite autenticazione
- Gestione delle operazioni nell'account
- Gestione del contenuto dei documenti

Il primo punto è analizzato ampliamente nella Fase II del progetto quindi in questa sezione saranno descritti soltanto gli altri due.

2.3.1 Gestione delle operazioni nell'account

Per operazioni sull'account si intendono tutte le attività di creazione ed eliminazione delle risorse dallo spazio di memorizzazione. Un prerequisito fondamentale per questa fase riguarda l'autenticazione. Questa operazione infatti è necessaria per associare un utente al proprio account dando modo di riferire ogni operazione eseguita dalla componente in maniera corretta, proteggendo l'accesso dagli altri utenti. Ogni account costituisce uno spazio di memorizzazione remoto paragonabile ad un grande file system all'interno del quale ogni documento costituisce una risorsa ed analogamente ad un file può essere creata ed eliminata. È quindi indispensabile che la componente definisca delle primitive che permettano di compiere queste operazioni rispettivamente per ogni tipologia di documento. Un'altra esigenza è quella di ottenere una lista dettagliata delle risorse presenti nell'account, sia per cercare una risorsa sia per avere un riferimento ad essa nel caso la si voglia utilizzare.

2.3.2 Gestione del contenuto

La gestione del contenuto dei documenti rappresenta la mansione principale di questa componente ed essenzialmente è la capacità di inserire dei dati all'interno di un documento rispettandone la struttura. Necessita ovviamente degli strumenti descritti sopra per creare fisicamente le risorse. Una delle esigenze principali sorte all'interno dell'azienda era quella di creare dei documenti a partire da informazioni estratte da una base di dati. Genericamente è possibile definire un data provider come la tipica sorgente delle informazioni. Questo strumento può essere organizzato in varie maniere a seconda della complessità e degli usi che un'applicazione vuole avere. In ogni caso l'accesso al data provider è controllato da un'interfaccia che permette di definire le interrogazioni ed ritornare i risultati. La natura dei dati uscenti da questo sistema sono

fortemente strutturati per agevolarne la memorizzazione all'interno del sistema e difficilmente si adattano ad essere inseriti all'interno di documenti il cui contenuto in generale è semi-strutturato. Per questo motivo la componente deve fornire delle primitive in grado di accedere alla struttura del documento per definire le modalità con cui i dati sono inseriti. Non solo, nel caso di un foglio di lavoro inoltre è possibile eseguire dei semplici calcoli sui dati attraverso la definizione di formule. Un tipico esempio rappresenta la creazione di una fattura. Supponiamo il data provider fornisca, come risultato di una interrogazione, la lista degli articoli con la rispettiva quantità e prezzo acquistati da un certo cliente in una certa data. Definendo cella per cella i vari valori per ogni articolo è facile immaginare come sia possibile calcolare il totale piuttosto che la percentuale d'iva, in maniera dinamica definendo una semplice formula all'interno della cella. Tutte queste operazioni diventano apprezzabili quando vi è l'associazione tra i dati e un modello astratto del documento che si vuole ottenere. Nel caso della fattura supponiamo il modello potrebbe definire la posizione delle celle che andranno a contenere i dettagli relativi all'intestazione piuttosto che la lista degli articoli, la definizione delle formule, la posizione dei risultati, ecc. in maniera dinamica rispetto ai dati. Un approccio di questo tipo permette di separare il contenuto dalla presentazione ma necessita di un algoritmo che, dato un modello e i dati, genera un documento utilizzando le primitive messe a disposizione dalla componente. Tuttavia in questo progetto mi sono limitato a sviluppare la componente mentre le considerazioni appena fatte rappresentano soltanto uno spunto per evoluzioni successive. Fin'ora ho discusso il caso in cui il flusso dei dati parta da un'applicazione mentre ora vorrei considerare il caso contrario. Il formato delle informazioni su cui i servizi Google si basano esclusivamente su XML. Questo implica che è possibile ottenere una rappresentazione dei documenti soltanto in questo formato, ma ciò non costituisce un limite in quanto XML rappresenta uno standard che sempre più si sta affermando nel web per l'interscambio di dati, e inoltre tutti gli applicativi web costruiti con gli strumenti *SitePainter* e *PortalStudio* integrano al loro interno componenti per un utilizzo immediato. Nel corso del documento saranno mostrati degli esempi in cui vi sarà proprio l'utilizzo di *XMLDataObject*, la componente descritta nel capitolo 1.2.4, per ottenere dei dati da una sorgente XML e utilizzarli all'interno dell'applicazione.

Riassumendo la componente sviluppata si limita a fornire dei metodi semplici per scrivere dati da un'applicazione in un documento come mostrato in Figura 4, e allo stesso modo il contrario. Tipicamente questi sono estratti da un data provider come dati strutturati e destrutturati in codice XML dalla componente medesima.

Figura 4

2.3.3 Requisiti architetturali

Intuitivamente l'integrabilità di un sistema software A rispetto ad un altro sistema software B è la possibilità di utilizzare in parte o totalmente i servizi e le funzionalità del primo direttamente dal secondo, ovvero B include le funzionalità A. Un sistema integrabile prende il nome di *third-party software component* e spesso viene utilizzato per garantire efficienza e qualità nello sviluppo di soluzioni personalizzate. Partendo da questo concetto ho cercato di raccogliere più informazioni possibili del sistema *Google Document*, per capire in che modo sarebbe stato possibile integrarlo in una generica applicazione realizzata tramite gli strumenti *SitePainter* e *PortalStudio*. Scartando l'idea di includere le funzionalità di *Google Document* tramite il riuso di codice sorgente adattandolo alle mie esigenze in quanto non disponibile, sono passato ad analizzare l'ipotesi di utilizzare l'API messa a disposizione da Google come interfaccia ai propri servizi. In base anche alle considerazioni fatte nel capitolo 2.2 questa libreria al livello più basso si basa sul protocollo http quindi uno dei problemi fondamentali che è sorto, dal momento che il contesto operativo riguarda le applicazioni Web, era la scelta delle modalità con cui è possibile interfacciarsi verso un sistema esterno. Una possibilità era quella di implementare una componente Javascript basata sull'architettura AJAX, quindi distribuita nel client, mentre un'altra possibilità era quella di implementare una componente Java distribuita quindi nel server. Nella Figura 5 è mostrata la diversità di questi due approcci, rispettivamente 1 e 2.

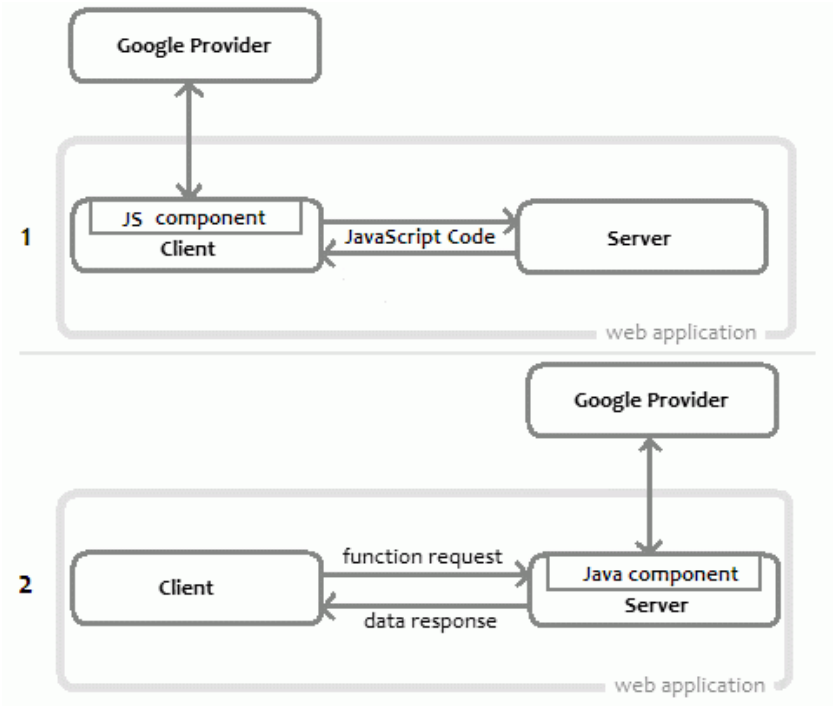


Figura 5

I vantaggi implementando la componente Java sono maggiori rispetto a di quelli che si hanno sviluppando la stessa componente tramite Javascript. Le differenze sono:

	Componente Javascript	Componente Java
Esecuzione	Sul client	Sul server
Sicurezza	Browser	JVM
Flusso dei dati	Server->Client->Google	Server->Google
Integrazione SitePainter	Si	Si

L'esecuzione sul client tipicamente richiede maggiori risorse a disposizione dell'utente, ma permette di evitare la gestione della multiutenza che si avrebbe nel caso dell'esecuzione sul server. Per quanto riguarda la sicurezza invece, il codice eseguito nel browser Web in generale è meno protetto da quello eseguito nella JVM. Ma la motivazione principale che mi ha spinto a scegliere il secondo approccio riguarda l'efficienza complessiva del sistema nella gestione del flusso dei dati, infatti l'architettura di un'applicazione web prevede che siano immagazzinati tipicamente in una o più basi di dati collocate all'interno del server. Dal momento che i dati devono essere inviati al Google Provider per la generazione dei documenti è assolutamente una scelta assurda richiedere che transitino per il client prima di essere utilizzati come avverrebbe nel caso la

componente fosse situata nel client ancor più nel caso la mole di dati fosse di notevoli dimensioni.
Definite le modalità con cui la componente deve operare è possibile analizzare il ruolo che occupa all'interno dell'applicazione. Come già anticipato è indispensabile che essa sia strutturata almeno su due livelli per il fatto che deve gestire una connessione sulla quale fornire dei servizi. Quindi per natura si interpone tra il client e il Google provider inoltrando le richieste dall'uno all'altro. Il client cioè non è mai in comunicazione diretta con il provider ma ogni informazione è mediata dalla componente che fornisce appunto l'accesso a tutti i servizi. Nei nostri esempi la componente deve essere inserita all'interno delle *servlet* che implementano la logica dell'applicazione.
Ricapitolando dal punto di vista architetturale i requisiti sono:

ID	Requisito
1.1a	La componente deve essere sviluppata in Java e deve essere situata nel server
1.2a	La componente deve essere strutturata su più livelli in cui il più basso si occupa della connessione verso il Google provider

2.3.4 Requisiti funzionali

Alla luce di quanto definito sopra sono passato a definire i requisiti funzionali che la componente deve soddisfare.

ID	Requisito
1.1.1f	Recupero della lista dettagliata dei documenti presenti all'interno di un account Google.
1.1.2f	Recupero dell'URL associato ad un documento.
1.1.3f	Recupero della chiave identificativa associata ad un documento.
1.1.4f	Recupero del titolo associato ad un documento.
1.2.1f	Creazione di un nuovo foglio di calcolo.
1.3.1f	Creazione di un nuovo documento di testo.
1.4.1f	Creazione di una nuova presentazione.
1.2.2f	Eliminazione di un foglio di calcolo.
1.3.2f	Eliminazione di un documento di testo.
1.4.2f	Eliminazione di una presentazione.

1.2.3f	Recupero del worksheet associato ad un foglio di calcolo.
1.2.4f	Modifica del contenuto di un worksheet.
1.2.5f	Modifica dei metadati associati ad un worksheet.
1.2.6f	Recupero del contenuto di un worksheet.

2.3.5 Integrazione in SitePainter

In seguito allo sviluppo della componente ho definito le modalità con cui questa può essere integrata in *SitePainter*. In effetti la componente sviluppata non è legata ad un contesto d'uso specifico e quindi può essere utilizzata in qualsiasi applicazione Java. Ho pensato quindi di incapsularla tramite lo strumento di *Library Editor*, descritto durante l'analisi dell'ambiente operativo nel capitolo 1.2.3, in una libreria permettendo così di essere utilizzata dall'interno del software di sviluppo. Quest'ultima rappresenta soltanto un involucro per la componente e quindi ne esporta le medesime funzionalità, ma soddisfa i seguenti requisiti:

ID	Requisito
1.1s	Tutti i metodi pubblici devono essere dichiarati statici.
1.2s	I parametri dei metodi devono essere di tipo stringa, numerico o booleano.
1.3s	Ogni valore ritornato deve essere di tipo stringa, numerico o booleano.

Questi requisiti sono dettati dalle modalità con cui *SitePainter* organizza le librerie. Inoltre l'utilizzo di tipi semplici come stringhe e numeri per i dati scambiati tra client e server permettono una serializzazione immediata, rispetto all'uso di articolate strutture dati.

2.4 Fase II

Durante questa fase mi sono occupato del problema dell'autenticazione, il requisito fondamentale per il funzionamento della componente.

2.4.1 Considerazioni sulla sicurezza

Durante questa fase, di impronta più teorica, ho potuto affrontare le problematiche di sicurezza a cui sono soggette le applicazioni Web. Descriverò quindi una panoramica generale sull'argomento e cercherò di indicare dove le nozioni assunte mi hanno permesso di analizzare in maniera critica il progetto.

In pochi anni il World Wide Web si è evoluto da un semplice fruitore di informazioni statiche ad uno di applicazioni interattive altamente

funzionali in grado di gestire informazioni sensibili ed eseguire operazioni piuttosto critiche. Le applicazioni Web, dette anche *Rich Internet Applications*, offrono alcuni sostanziali vantaggi rispetto alle applicazioni tradizionali sebbene l'esecuzione in un browser Web possa essere più limitante e lo sviluppo più intricato. Tra questi è possibile riscontrare:

- Installazione assente così come le spese generali per l'aggiornamento e la distribuzione.
- Aggiornamento alle nuove versioni trasparente agli utenti.
- Utilizzo da qualsiasi client collegato alla rete.
- Indipendenza dalla piattaforma e dal sistema operativo usato.
- Minore esposizione ad infezioni virali.

Questo contribuisce notevolmente a mantenere un *Total Cost of Ownership*, cioè l'indice utilizzato per calcolare tutti i costi relativi al ciclo di vita di un software, estremamente basso.

Per quanto riguarda la portabilità di un'applicazione desktop questa è influita dal linguaggio di programmazione, mentre per una Web dal formato di presentazione dei dati, poiché l'ambiente d'esecuzione è nel primo caso il sistema operativo, mentre nel secondo il browser Web. Uno dei primi approcci nello sviluppo di applicazioni Web è stato lanciato da Sun nell'anno 1995 attraverso le Applet Java. Inizialmente furono molto utilizzate in quanto potevano essere incluse come oggetti nelle pagine Web ed eseguite dalla JVM contenuta nel browser. Nessun'altra tecnologia permetteva di realizzare funzionalità e interattività tipiche delle Applet, quindi spesso erano utilizzate anche per assolvere dei compiti molto banali. Il prezzo da pagare per l'utilizzo delle Applet Java consiste nell'avvio della JVM, fin troppo pesante per molti sistemi e dispositivi portatili. Inoltre è frequente una discrepanza di *rendering* grafico dell'applicazione tra le diverse implementazioni della macchina virtuale che i diversi browser contengono. Il vero punto forte di questa tecnologia riguarda la sicurezza. Ogni applicazione Java quindi di fatto anche le Applet implementa una sofisticata politica di sicurezza a più livelli. La piattaforma stessa implementa numerose protezioni sia in fase di compilazione sia di esecuzione, tra queste vi è:

- Forte tipizzazione e controllo dei dati.
- Gestione automatica della memoria.
- Verifica del *bytecode* e caricamento sicuro delle classi tramite firme e certificati digitali.

Ma come già detto questo si paga nelle prestazioni. Con la nascita di un leggero ma efficace linguaggio di *scripting* come Javascript e l'arrivo di nuovi standard per il Web, l'utilizzo delle Applet è venuto sempre meno e riservato soltanto a gestire comunicazione asincrone tra client e server. La stessa azienda Zucchetti che fino all'anno 1999 le utilizzava nello sviluppo delle proprie applicazioni si accorse delle potenzialità di Javascript sia per manipolare il contenuto di una pagina sia per gestire le connessioni asincrone con il server. L'essenza di AJAX è stata presto assimilata e molti dei problemi che c'erano con l'utilizzo della tecnologia precedente sparirono. Oggi l'azienda è concentrata nello sviluppo automatizzato di applicazioni Web gestionali fortemente interattive con l'obbiettivo di cercare soluzioni sempre migliori per risolvere problematiche intrinsecamente complicate, come ad esempio la sicurezza: il vero tallone d'Achille per i moderni sistemi Web. Questi contengono e controllano dati e risorse, pertanto devono essere progettati ed implementati per proteggerli. Nessuna applicazione può essere considerata veramente sicura senza che gli elementi che costituiscono l'ambiente operativo sono sicuri loro stessi. E' quindi indispensabile individuare l'esposizione ad attacchi causati dall'ambiente operativo, in particolare gli utenti. Una regola fondamentale per controllare il problema nella sua globalità è non fidarsi mai dei dati provenienti dagli utenti, e mai fare ipotesi circa i limiti delle tecnologie a disposizione e le azioni che possono compiere tramite l'uso di queste. Queste semplici ma efficaci indicazioni presentano il problema essenziale, ossia che l'utente ha la possibilità di inviare input completamente arbitrari all'applicazione lato server interferendo con la sua logica per ottenere funzionalità e informazioni non autorizzate. Questo problema si può manifestare in molti modi. In generale l'utente ha o uscente dal server, sia essa un parametro di richiesta piuttosto che un cookie piuttosto che un'intestazione HTTP. Ma può addirittura inviare richieste disordinate o parametri incompleti in istanti diversi, in maniera imprevedibile. Esistono infatti molti strumenti che operano al di fuori del browser, per creare situazioni inaspettate. Un altro problema è dovuto al fatto che l'utilizzo di nuove tecnologie apre la strada ad un gran numero di falle non sempre risolte dagli sviluppatori di browser Web in maniera tempestiva.

Dopo questa introduzione descrivo brevemente i dettagli relativi alla sicurezza in Javascript, il linguaggio client-side utilizzato dalle applicazioni sviluppate durante questo progetto.
La differenza che sussiste tra un linguaggio di programmazione e uno di *scripting* è che con il primo si ha una netta partizione tra i dati e il codice del programma, implicata dal processo di compilazione. Idealmente non è possibile che un utente esegua del codice dopo che questa sia già stata generata e eseguita. Mentre nel Web ciò non avviene, è possibile mescolare dati e codice creando enormi problemi di sicurezza. La motivazione sta nel fatto che il server e il client, funzionano su due livelli applicativi differenti. Il server genera dinamicamente contenuti che il client è in grado di interpretare e visualizzare. Questi contenuti tuttavia possono dipendere da dati e informazioni inviati dal client stesso o da altri client in momenti diversi, innescando un meccanismo circolare in grado talvolta di compromettere il sistema. I browser, con lo scopo di contenere questo rischio, utilizzano due restrizioni principali. In primo luogo eseguono gli script in una *sandbox*, ovvero quel meccanismo di sicurezza che permette l'esecuzione in un ambiente confinato in cui si limitano le funzionalità ad un contesto prettamente Web. In secondo luogo applicano una politica detta *Same Origin Policy*, cioè la più importante misura di sicurezza per lo *scripting* client-side. La maggior parte dei bug di sicurezza provengono da violazioni di questo principio. Essa previene che un documento o script proveniente da una certa origine possa ottenere o modificare proprietà di un documento con una origine diversa. L'origine di una risorsa è definita usando la notazione URL, cioè nome del dominio, protocollo e porta quindi due pagine appartengono alla stessa origine se e solo se questi tre valori sono identici. Senza questa protezione, una pagina maliziosa potrebbe interferire compromettendo la confidenzialità o l'integrità di un'altra pagina Web. In generale la sicurezza rappresenta un problema talmente vasto che va controllato per parti. Nonostante la grande varietà di applicazioni Web, praticamente tutte utilizzano lo stesso modello di sicurezza. I meccanismi che lo compongono rappresentano una difesa di base contro utenti malintenzionati e proteggono dalle vulnerabilità più diffuse. Di queste componenti, la gestione dell'accesso degli utenti e la gestione dell'input sono i più importanti. Difetti in questi meccanismi spesso portano ad una compromissione totale dell'applicazione.

La gestione dell'accesso rappresenta il requisito principale che un'applicazione Web dovrebbe possedere, infatti permette di distinguere gli utenti e di confinare le operazioni e i dati che essi possono utilizzare. Questa operazione è costituita da due parti fondamentali, rispettivamente l'autenticazione e la gestione delle sessioni. La prima è ampliamente analizzata alla fine di questa sezione perché è stata applicata relativamente a questo progetto, mentre la seconda è dovuta al fatto che il protocollo HTTP, sul quale si basano la maggior parte delle comunicazioni tramite Internet, è *stateless*. Tipicamente un utente accede alle varie pagine e funzioni di una applicazione facendo una serie di richieste HTTP dal proprio browser. Allo stesso tempo, il server riceve queste richieste assieme a quelle provenienti da altri utenti. Quindi è necessario identificarle ed organizzarle in sessioni distinte, una per ogni utente. Questo meccanismo può essere implementato in molti modi, ma il principio fondamentale è che per ciascuna sessione di ciascun utente l'applicazione deve rilasciare un identificativo univoco, generato e memorizzato dal server in una struttura dati apposita. Per tutta la durata della sessione ogni richiesta deve contenere al suo interno l'identificativo, così da essere correttamente associata ad un utente. I Cookie HTTP sono il metodo più utilizzato per mantenere una informazione sul client anche se ad esempio è possibile utilizzare campi invisibili all'interno delle pagine. Il Cookie non è altro che un pezzo di informazione aggiunta nell'intestazione delle richieste HTTP automaticamente dal browser, e mantenuto in un file. La creazione è richiesta dal server tramite l'impostazione del campo Set-Cookie dell'header HTTP. L'intero meccanismo è trasparente all'utente. In termini di sicurezza, il meccanismo delle sessioni dipende direttamente dall'affidabilità e univocità degli identificativi di sessione, in particolare dalla lunghezza e dalla casualità con cui sono generati. Avere a disposizione un identificativo talvolta è sufficiente per ottenere l'accesso all'applicazione con gli stessi diritti dell'utente che ha aperto la sessione.

Un altro problema fondamentale riguarda la gestione dell'input, già accennato precedentemente. In generale è possibile osservare un fenomeno di nomadismo del codice che non segue il flusso normale dei dati cioè dal server al client, ma ha la possibilità di risalire, mescolato ai dati stessi, nel verso opposto e quindi di propagarsi per tutta l'applicazione. È indispensabile creare un filtro che impedisce al codice pericoloso di spostarsi all'interno dell'applicazione e quindi di

essere distribuito indirettamente a tutti gli utenti. Una validazione dell'input sistematica rappresenta la difesa necessaria a questo problema anche se costituisce un'operazione piuttosto pesante perché da farsi in ogni punto dell'applicazione potenzialmente modificabile dall'utente.

Esistono svariate tecniche per fare questo a seconda del *trade-off* che si vuole garantire tra sicurezza e velocità d'esecuzione. Inoltre è da considerare che esiste una grande varietà di input possibili, sia generati dagli utenti sia generati dall'applicazione, quindi fissare una codifica standard alla quale fare riferimento evita eventuali equivoci. Un primo approccio detto di "Reject Known Bad" impiega una *black-list* contenente una serie di stringhe o modelli noti per essere utilizzati negli attacchi. Il meccanismo di convalida blocca tutto ciò che corrisponda alla black-list e permette il resto. Non è molto efficace in quanto è necessario prevedere tutti gli input pericolosi in base anche alle diverse codifiche che possono assumere. L'approccio complementare detto "Accept Known Good" utilizza invece una *white-list* contenente una serie di stringhe, pattern, criteri, espressioni regolari che garantiscono l'affidabilità dei dati. È molto più flessibile del primo, ma esistono molte situazioni in cui non è applicabile. Un approccio più sofisticato ed adottato anche dall'azienda all'interno dei propri applicativi è chiamato Sanitizzazzazione, in cui si riconosce la necessità di accettare talvolta dati che potrebbero essere pericolosi. Invece di rifiutarli, l'applicazione li modifica in vari modi per garantire che non vengano mai interpretati e quindi eseguiti dal browser. I caratteri potenzialmente dannosi possono essere rimossi oppure opportunamente codificati. Approcci basati sulla sanitizzazione dei dati sono molto efficaci, e possono essere utilizzati come una soluzione generale al problema della validazione. Altre tecniche analizzano l'input non a livello sintattico ma a livello semantico. Questi sono meno utilizzati poiché necessitano di algoritmi sofisticati e computazionalmente onerosi.

I metodi appena descritti costituiscono le componenti principali di un modello più generale di sicurezza che utilizza il concetto di "Boundary Validation" (validazione sul confine). Si basa sul principio in cui i dati devono essere sempre validati prima di entrare ed uscire dalle componenti del programma che li utilizzano. E' errato infatti assumere che dopo una validazione i dati preservino la proprietà di affidabilità perché ad esempio se un input viene utilizzato per una

serie di operazioni collegate, dove l'output di una costituisce l'input per un'altra e così via, e visto che alcuni filtri applicano delle trasformazioni sui dati allora un utente malintenzionato potrebbe essere in grado di sfruttarle per ingannare l'intero processo di validazione. Inoltre può capitare che la difesa impiegata contro una tipologia di attacchi sia incompatibile con quella impiegata per altre tipologie. Il modello di "Boundary Validation" rappresenta quindi una soluzione efficiente per gestire la sicurezza dell'input in tutte le operazioni compiute all'interno e all'esterno dell'applicazione.

2.4.2 L'autenticazione

In generale l'autenticazione è l'operazione con cui l'utente dimostra la propria identità, concettualmente è un meccanismo di sicurezza molto semplice talvolta però necessità di protocolli molto complessi al fine di garantire un adeguato livello di sicurezza, soprattutto quando si ha a che fare con applicazioni accessibili dal Web. Fondamentalmente Google permette di gestire l'autenticazione in due modi diversi. Un primo modo implica l'immissione delle credenziali utente attraverso metodi forniti dalle sue API, mentre un secondo modo implica l'immissione di un *token* ottenuto durante l'esecuzione del protocollo *AuthSub*. In questa sezione sarà approfondito l'utilizzo del secondo approccio in quanto considerato più affidabile e consigliato dalla stessa Google agli sviluppatori di applicazioni Web. A questo punto è già possibile identificare quelli che sono i requisiti a cui la componente deve rispondere, in generale offrire dei metodi che permettano la gestione di entrambe le soluzioni.

ID	Requisito
2.1	Autenticazione tramite username e password Google memorizzati in una base di dati.
2.2	Autenticazione tramite il protocollo AuthSub.

Le differenze principali che sussistono tra le due soluzioni sono legate all'uso che se ne fa della componente. Nel nostro caso visto che la componente è collocata nel server dell'applicazione è naturale vedere come nel primo approccio le credenziali debbano raggiungere la componente, quindi il server, attraverso l'uso di un protocollo sicuro per il trasferimento e un'interfaccia utente adeguata all'immissione, mentre nel secondo caso, come sarà descritto successivamente, no. Piuttosto che richiedere l'inserimento delle credenziali ad ogni accesso talvolta è conveniente che queste siano

ospitate in una base di dati, quindi all'interno dell'applicazione, ma questo implica l'adozione di elevati controlli di sicurezza e di protezione a carico del sistema di memorizzazione. Per cui la sicurezza del sistema globale è pari alla sicurezza che presta l'applicazione verso gli utenti. Per quanto riguarda l'utilizzo del protocollo *AuthSub* tutto questo non avviene. L'autenticazione è gestita attraverso un provider di autenticazione proprietario di Google. L'evidente vantaggio che fornisce l'uso di quest'ultimo consiste nella robustezza dovuta ad un'interazione chiara e ben definita tra l'applicazione l'utente e Google, quindi la sicurezza dipende soltanto dall'affidabilità del protocollo in questione. L'unica cosa di cui necessita *AuthSub* è di un'infrastruttura in grado di gestire le transazioni tra l'utente, l'applicazione e il provider Google. Sperimentalmente in questo progetto ho sviluppato le componenti indispensabili ad utilizzare questo protocollo e saranno descritte nel corso del capitolo 3.2.2. Passo ora a descrivere Il funzionamento nel dettaglio.

L'azione di autenticazione è strettamente correlata alla gestione della sessione in quanto entrambe condividono lo scambio di una informazione che viene rilasciata da un sistema per identificare le azioni di un utente rispetto a quel sistema. Il protocollo *AuthSub* nasce dall'idea di consentire l'accesso ai servizi Google da applicazioni Web di terze parti senza costringere queste a gestire direttamente le credenziali. La tecnica con la quale opera il protocollo consiste nello scambio di un identificativo rilasciato ad un utente nel momento dell'autenticazione ed associato ad esso fino al termine. Lo scenario tipico consiste come in Figura 6 di tre parti, rispettivamente l'utente o meglio il browser in cui opera, la *Web-application* e il *Google Account Authentication*. Il protocollo *AuthSub* coordina queste tre parti al fine di garantire la sicurezza.

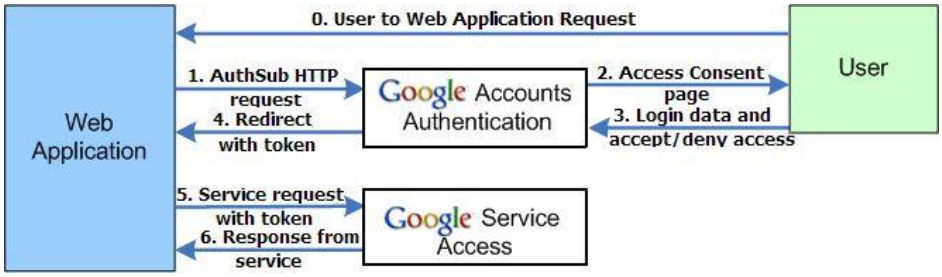


Figura 6

Le fasi che compongono il protocollo sono:

0. L’utente richiede all’applicazione Web un servizio per il quale è richiesta l’autenticazione presso il Google Provider, ad esempio *Google Document*.
1. L’applicazione Web contatta il *Google Accounts Authentication* Provider per reindirizzare l’utente alla pagina di autenticazione.
2. L’utente ottiene una pagina in cui consente l’accesso dell’applicazione al servizio Google.
3. Quindi inserisce le credenziali e conferma l’autenticazione.
4. Il *Google Account Authentication* verifica le credenziali e se corrette reindirizza l’utente all’applicazione Web restituendo un *token* identificativo.
5. A questo punto ogni richiesta dell’applicazione verso il Provider Google può essere autenticata e il protocollo termina.
6. Il servizio risponde con i dati richiesti.

Dal punto di vista logico l’approccio è molto immediato ma è necessario precisare alcune considerazioni.

Innanzi tutto si vede come nei passaggi 2 e 3 vi sia un’uscita temporanea dall’applicazione verso il portale di autenticazione Google che rappresenta una situazione piuttosto scomoda da gestire quando si utilizza un layout della pagina molto ristretto come quello di una *Portlet*. Per questo nel mio caso ho preferito aprire questa sezione in una pagina separata andando leggermente a discapito dell’usabilità.

Un'altra considerazione riguarda il *token* identificativo. Ogni qualvolta viene rilasciato dal *Google Accounts Authentication Provider* nel passaggio 4 questo è ad uso singolo, ovvero è valido per una volta soltanto e poi scade automaticamente. Cosicché è necessario provvedere alla conversione ad un *token* di sessione caratterizzato da una durata illimitata. Inoltre un *token* possiede una visibilità o scope per un servizio specifico, cioè al momento dell’autenticazione è necessario specificare il servizio Google con il quale s’intende comunicare. Un problema sorge nel momento in cui sia necessario utilizzare più servizi in contemporanea, come nel nostro caso Spreadsheet e Document appartengono a due scope diversi. Questo rappresenta un grosso limite che sarebbe superato solo nel caso in cui Google mettesse a disposizione *token* a scope multiplo, cioè validi per più servizi contemporaneamente.

Riassumendo, le differenze nell'uso di *AuthSub* piuttosto che far pervenire le credenziali in maniera diretta alla componente si possono riassumere nei seguenti tre punti:

- Infrastruttura necessaria
- Sicurezza ed affidabilità
- Facilità d'uso dell'applicazione

L'infrastruttura necessaria nel primo caso è determinata dalla necessità di gestire il *token* e sarà analizzata nella sezione successiva, nel secondo caso è necessario una tabella nella base di dati che mantenga le credenziali in chiaro ed un sistema in grado di associare gli utenti ai propri account correttamente. In caso di molti utenti diventa problematico gestire queste informazioni in un sistema distribuito e accessibile soltanto dall'amministratore del sistema, quindi ne rappresenta un limite. Allo stesso modo anche la sicurezza e l'affidabilità sono condizionate dalle stesse problematiche. La disponibilità da parte di un malintenzionato di un *token* di sessione circoscrive molto il campo d'azione in quanto la validità è molto ridotta rispetto a quella di una credenziale. L'usabilità invece va leggermente a sfavore del protocollo poiché è necessario gestire la visualizzazione di una pagina di login che non dipende direttamente dall'applicazione.

2.5 Scelte operative

Avendo svolto il lavoro in maniera prevalentemente autonoma ho scelto di adottare un approccio che si discosta da quello ingegneristico che prevede una forte presenza di documentazione, in quanto gran parte del lavoro è stato di analisi. La componente prodotta non costituisce la parte di un progetto quindi l'utilizzo è completamente indipendente. Ho trascorso molto tempo cercando di documentarmi alla ricerca di informazioni che mi permettessero di implementare uno strumento semplice ma funzionale. Comunque alla fine di tutto è stata rilasciata all'azienda la documentazione necessaria da permetterne un utilizzo immediato.

3 Definizione delle attività

3.1 Fase I

In questa sezione sono descritti i passi compiuti durante la progettazione e lo sviluppo della componente Java, e sono mostrati alcuni esempi d'uso della componente.

3.1.1 Analisi architetturale

Per la progettazione e lo sviluppo della componente ho utilizzato esclusivamente l'IDE *NetBeans* con i relativi plug-in per UML.
L'idea di una componente nasce dal momento in cui, avendo capito il funzionamento del protocollo di comunicazione utilizzato da Google per i propri servizi, ho pensato di classificare le tre tipologie di documento come entità separate in modo tale da ridurre al minimo la dipendenza tra le classi e quindi di agevolare le modifiche alle singole parti. L'architettura globale è definita su due livelli come mostrato in Figura 7.

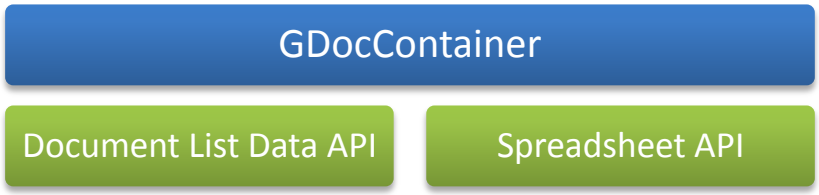


Figura 7

Sotto vi è il livello che controlla la comunicazione e lo scambio dei dati costituito dalle librerie *Document List Data API* e *Spreadsheet API*, mentre sopra la gestione dei documenti e delle altre funzionalità. Inizialmente lo sviluppo era partito dal livello più basso ovvero con la creazione di una classe per la comunicazione con il Google Provider attraverso il protocollo HTTP, ma subito abbandonata a causa delle difficoltà incontrate nella creazione delle risorse secondo il formato *Atom*. L'operazione di costruzione dei *feed Atom* ed i relativi nomi spazio era scarsamente documentata e comunque rimpiazzabile dall'uso delle librerie citate prima e descritte nel capitolo 2.2. L'idea sulla quale è stata sviluppata la componente si basa sulla rappresentazione di un documento come un oggetto, quindi una classe principale è in grado di raccogliere tutti i documenti classificati per tipologia. La struttura dati di base è quella tipica di un contenitore, dalla quale deriva la scelta del nome *GDocContainer*.

Tutte le operazioni effettuate sul contenitore sono immediatamente visibili anche dall'applicazione *Documents* di Google in quanto ogni oggetto è associato, tramite un riferimento remoto, ad un documento presente nell'account. Il contenitore ha inoltre due stati a seconda che sia stata effettuata l'autenticazione oppure no. Nel secondo caso infatti la componente è disattivata, cioè nessun metodo è funzionante poiché non è associato a un account. Una istanza del contenitore può essere associata contemporaneamente ad uno e un solo utente, ma è permesso avere più istanze per gestire più utenti. Dal punto di vista architetturale aggrega le due librerie: *Document List Data API* e *Spreadsheet API* con lo scopo di gestire sia operazioni di creazione ed eliminazione dei documenti tramite la prima sia per gestire nel dettaglio ciò che riguarda gli spreadsheet tramite la seconda. La necessità di utilizzarle entrambe nasce dal fatto che la prima è di utilizzo generale per l'organizzazione dei documenti nell'account e non fornisce alcun metodo specifico per definire il contenuto di un documento, mentre la seconda è specializzata nella gestione degli spreadsheet. Le tipologie di documento gestite quindi sono tre, rispettivamente:

- Documento di testo (Write)
- Presentazione (Presentation)
- Foglio di calcolo (Spreadsheet)

La mancanza di alcune funzionalità implementate dalle API ha determinato anche una variazione sui metodi messi a disposizione dalla componente per ogni tipologia di documento. In assoluto gli spreadsheet forniscono un controllo completo in quanto utilizzano una libreria apposita, mentre negli altri casi il controllo è solo parziale. Per nessuna delle tre tipologie ad esempio vi è un controllo sulla formattazione del testo. Non è escluso comunque che future versioni o implementazione di nuove librerie possano colmare questo gap. Precisamente le funzionalità a disposizione sono riassunte nella tabella.

Tipologia	Creazione logica	Creazione da File	Eliminazione	Controllo del contenuto
Spreadsheet	SI	SI	SI	SI
Write	SI	SI	SI	NO
Presentation	NO	SI	SI	NO

Per creazione logica s'intende la possibilità di creare un'istanza vuota di un documento.

3.1.2 Specifica tecnica

Tornando a ciò che riguarda la componente, l'architettura globale è mostrata in Figura 8.

Figura 8

GDocContainer è la classe principale, e al suo interno definisce le seguenti classi innestate:

- GSpreadContainer
- GWriteContainer
- GPresentationContainer

Rappresentano anch'esse dei contenitori rispettivamente per le tre tipologie di oggetti contenuti. Definiamo queste classi di secondo livello perché sono fortemente dipendenti dalla classe principale a causa dei dati che le accomunano. Esiste inoltre un terzo livello costituito dalle classi GSpread, GWriteDocument, GPresentation che forniscono una rappresentazione per i tipi di documenti descritti prima. È possibile notare come questa gerarchia sia facilmente espandibile nel caso vi sia la necessità di aggiungere altri tipi di documento oppure modificare quelli già esistenti infatti l'accoppiamento tra le classi è orientato verticalmente rispetto allo schema di Figura 8 e non orizzontalmente. Una piccola nota deriva nel caso degli spreadsheet che utilizzano un quarto livello per descrivere un oggetto di tipo *worksheet*. Un *worksheet* rappresenta il foglio di lavoro vero e proprio all'interno del documento, tipicamente organizzato a celle. È consentito definirne più di uno all'interno dello stesso documento.

La motivazione sulla scelta dell'utilizzo di classi innestate deriva dalla possibilità di aggregarle mantenendo sempre un riferimento alla classe principale e quindi permettendo l'utilizzo anche dei metodi privati. Quando le classi innestate sono dichiarate private, come nel nostro caso, non è permessa la creazione di istanze al di fuori della classe principale ed è necessario definire delle interfacce pubbliche che permettano di utilizzarne i metodi. Questa tecnica assicura l'integrità tra gli oggetti per il fatto che ognuno di questi è stato

creato correttamente. Ad ogni classe innestata quindi è associato un'interfaccia pubblica.

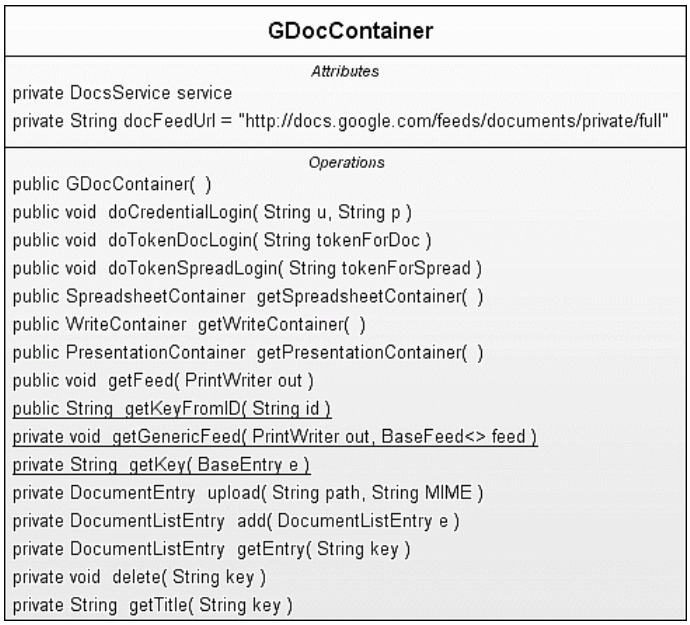


Figura 9

GDocContainer è la classe principale la cui interfaccia è mostrata in Figura 9.

I primi tre metodi permettono di ottenere un oggetto delle rispettive classi interne e sono:

- public SpreadsheetContainer getSpreadsheetContainer()
- public WriteContainer getWriteContainer()
- public PresentationContainer getPresentationContainer()

I seguenti metodi permettono invece di gestire l'autenticazione nelle due maniere possibili descritte durante la Fase II, e sono:

- public void doCredentialLogin(String u, String p)

  Effettua l'autenticazione fornendo la coppia di valori username e password rispettivi ad un account Google esistente.
- public void doTokenDocLogin(String tokenForDoc)

  Effettua l'autenticazione fornendo un token di sessione corretto con lo scope relativo al servizio Documents.
- public void doTokenSpreadLogin(String tokenForSpread)

  Effettua l'eutenticazione fornendo un token di sessione corretto con lo scope relative al servizio Spreadsheet.

Infine i metodi successivi sono di utilità generale:

- public void getFeed(PrintWriter out)

  Permette di ottenere una lista dettagliata di tutti i documenti presenti all'interno dell'account sottoforma di feed.
- public static String getKeyFromID(String id)

Permette di ottenere la chiave univoca del documento dato l'identificativo.

I restanti metodi sono privati quindi di utilità per le classi interne e sono utilizzati per la creazione, l'eliminazione, il caricamento dei singoli documenti attraverso la Document List Data API.

A secondo livello troviamo la classe GSpreadContainer di Figura 10.

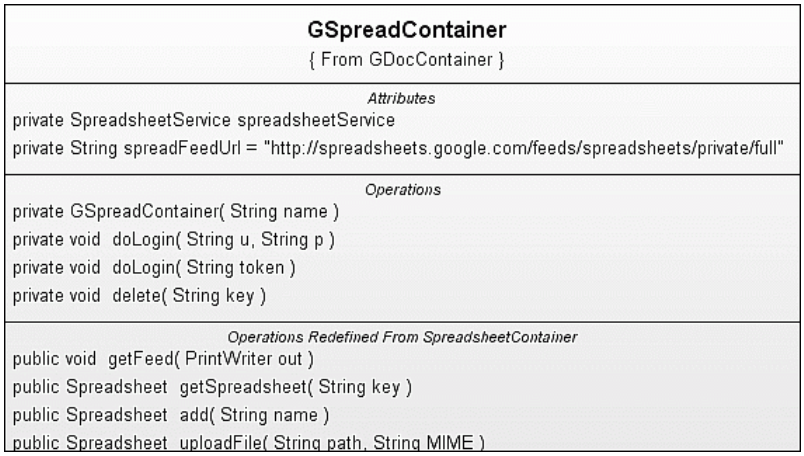


Figura 10

La classe GSpreadContainer è privata, ma implementa l'interfaccia SpreadContainer e fornisce l'implementazione dei metodi:

- public void getFeed(PrintWriter out)

  Consente di ottenere la lista dettagliata degli spreadsheet.

- public Spreadsheet getSpreadsheet(String key)

  Ottiene un oggetto spreadsheet identificato dalla rispettiva chiave.

- public Spreadsheet add(String name)

  Permette di aggiungere uno spreadsheet specificandone il nome, ritorna l'oggetto associato al documento creato.

- public Spreadsheet uploadFile(String path, String MIME)

  Permette di effettuare l'upload di un file locale specificandone il cammino e il tipo, ritorna l'oggetto associato al documento creato.

Sempre a secondo livello si trova la classe GWriteContainer di Figura 11.

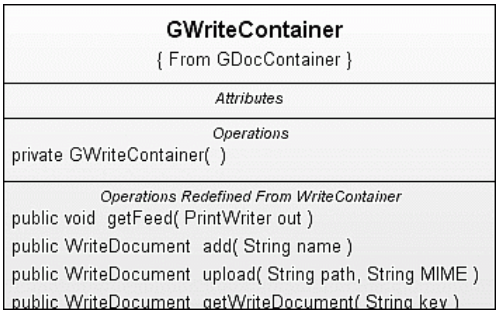


Figura 11

Anche questa è privata ma implementa l'interfaccia WritContainer e fornisce i seguenti metodi:

- public void getFeed(PrintWriter out)

  Consente di ottenere una lista dettagliata dei documenti di testo tramite un feed.

- public WriteDocument add(String name)

  Permette di creare un documento di testo vuoto specificandone il titolo, ritorna l'oggetto associato al documento creato.

- public WriteDocument upload (String path, String MIME)

  Permette di effettuare l'upload di un file specificandone il cammino e il tipo, ritorna l'oggetto associato al documento creato.

- public WriteDocument getWriteDocument(String key)

  Ottiene un oggetto associato al documento identificato dalla chiave corrispondente.

Infine l'ultima classe è GPresentationContainer di Figura 12.

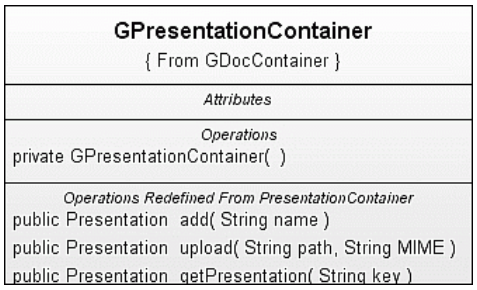


Figura 12

Questa classe implementa l'interfaccia PresentationContainer e differisce leggermente dalle altre due classi in quanto alcuni metodi sono assenti. In particolare l'attuale versione non prevede la creazione di presentazioni e neppure la possibilità di elencare quelle presenti nell'account, operazione che può essere fatta però dal metodo rispettivo della classe principale. I restanti metodi sono analoghi a quelli delle precedenti classi e sono:

- public Presentation upload (String path, String MIME)

- public Presentation getPresentation(String key)

Associata ad ogni classe contenitore vi è la corrispondente classe documento. Ognuna di queste definisce i metodi che permettono la manipolazione del documento stesso e sono pressoché simili. Nell'ordine c'è la classe innestata e privata GSpread in Figura 13 che rappresenta uno spreadsheet e implementa l'interfaccia Spreadsheet.

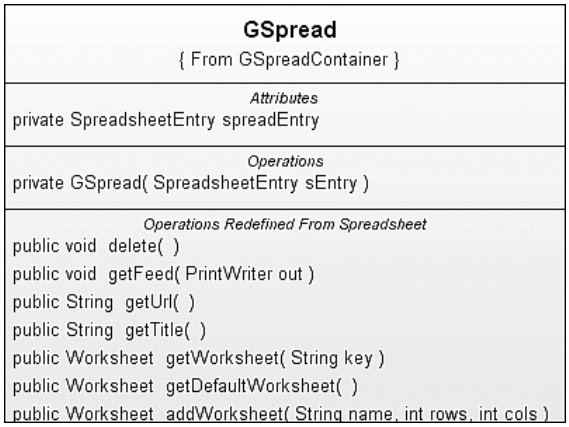


Figura 13

Pubblica i seguenti metodi:

- public void delete()

  Elimina lo spreadsheet associato all'oggetto sul quale questo metodo è chiamato.

- public void getFeed(PrintWriter out)

  Ottiene un feed contenente i dettagli relativi al documento.

- public String getUrl()

  Ritorna l'URL associato allo spreadsheet.

- public String getTitle()

  Ritorna il titolo associato allo spreadsheet.

- public Worksheet getWorksheet(String key)

  Ottiene un oggetto associato al worksheet corrispondente alla chiave indicata.

- public Worksheet getDefaultWorksheet()

  Ottiene un oggetto associato al worksheet di default

- public Worksheet addWorksheet(String name, int rows, int cols)

  Permette di aggiungere un *worksheet* all'interno dello spreadsheet specificandone il nome e la dimensione in righe e colonne.

Successivamente la classe GWriteDocument in Figura 14, anche questa innestata e privata che rappresenta un documento di testo, implementa l'interfaccia WriteDocument.

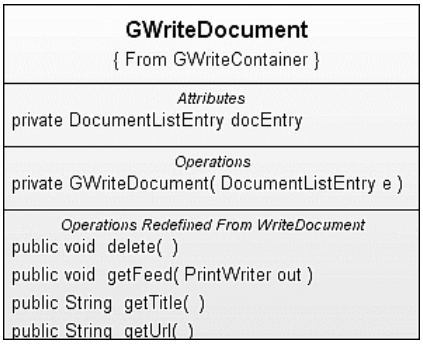


Figura 14

Implementa i seguenti metodi:

- public void delete()

  Elimina il documento associato all'oggetto sul quale questo metodo è chiamato.

- public void getFeed(PrintWriter out)

  Ottiene un feed contenente i dettagli relativi al documento.

- public String getTitle( )

  Ottiene il titolo assoiato al documento.

- public String getUrl()

  Ottiene l'URL associate al documento.

Ed infine la classe GPresentation in Figura 15, innestata privata implementa l'interfaccia Presentation.

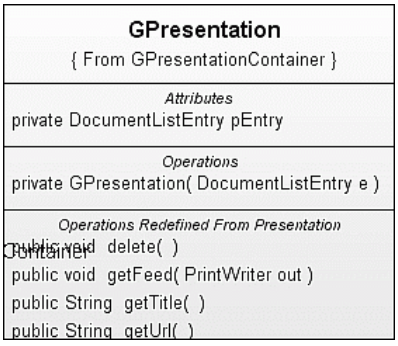


Figura 15

Implementa i medesimi metodi sopra descritti ovvero:

- public void delete()
- public void getFeed(PrintWriter out)
- public String getTitle( )
- public String getUrl()

Come ultima classe rimane quella associata ad un *worksheet* Figura 16. La classe GWorksheet è privata ed innestata nella classe GSpread, implementa l'interfaccia Worksheet.

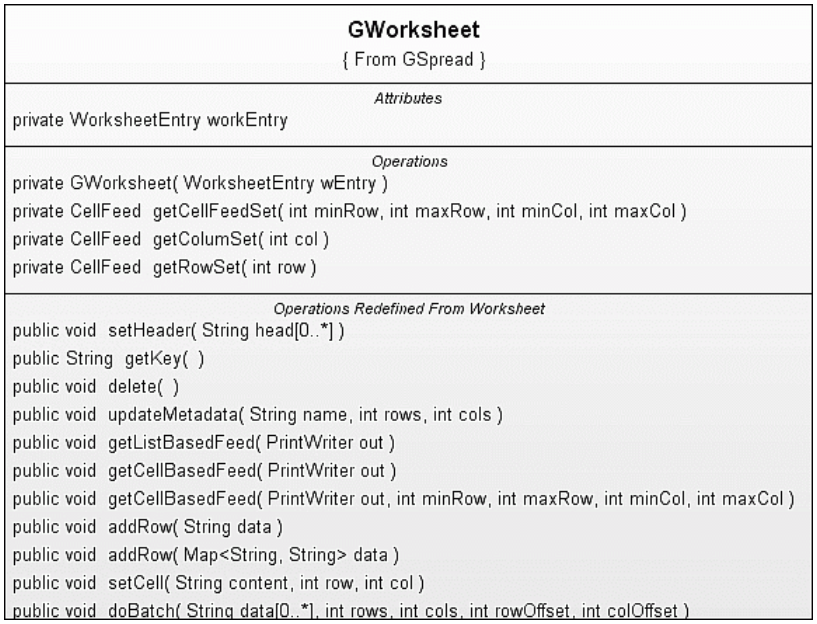


Figura 16

È la classe più articolata in quanto manipola direttamente il contenuto del foglio di lavoro rappresentato dall'insieme di celle. Implementa i seguenti metodi:

- public void setHeader(List<String> head)

  Permette di impostare un'intestazione al worksheet, costituito dalle celle che compongono la prima riga. Ognuna rappresenta un nome identificativo per la colonna corrispondente e viene utilizzato durante l'inserimento per riga con il metodo addRow.

- public String getKey()

  Ottiene la chiave identificativa corrispondente.

- public void delete()

  Elimina il worksheet corrispondente all'oggetto sul quale viene eseguito il metodo.

- public void updateMetadata(String name, int rows, int cols)

  Modifica i metadati costituiti dal nome e la dimensione del worksheet.

- public void getListBasedFeed(PrintWriter out)

  Ottiene un feed del contenuto del worksheet in cui ogni entry identifica una riga.

- public void getCellBasedFeed(PrintWriter out)

  Ottiene un feed del contenuto del worksheet in cui ogni entry identifica una cella.

- public void getCellBasedFeed(PrintWriter out, int minRow, int maxRow, int minCol, int maxCol)

Ottiene il feed di una porzione del contenuto del worksheet in cui ogni entry identifica una cella.

- public void addRow(String data)

Aggiunge una riga al worksheet dove i valori sono passati tramite una stringa secondo la notazione "columnName1=cellContent1, columnName2=cellContent2" ecc. I vari columnNameX sono i valori definiti tramite il metodo setHeader. La chiamata di questo metodo funziona soltanto dopo aver impostato un header valido. Non è possibile l'inserimento di formule.

- public void addRow(Map<String,String> data)

Identico al metodo omonimo con la differenza che il contenuto è mappato. Non è possibile l'inserimento di formule.

- public void setCell(String content , int row, int col)

Imposta il contenuto di una singola cella identificata dalle sue coordinate. E' possibile inserire formule.

- public void doBatch(List<String> data , int rows, int cols, int rowOffset, int colOffset)

È un metodo efficiente per inserimenti multipli di dati all'interno dello stesso worksheet. È necessario indicare la dimensione della regione di inserimento e quanto essa si discosta dall'origine. Per ogni chiamata a questo metodo non è consentito superare una dimensione di regione di 100 righe per 20 colonne. E' possibile inserire formule.

La conformazione finale della componente dal punto di vista architetturale è riassunto nel seguente diagramma delle classi.

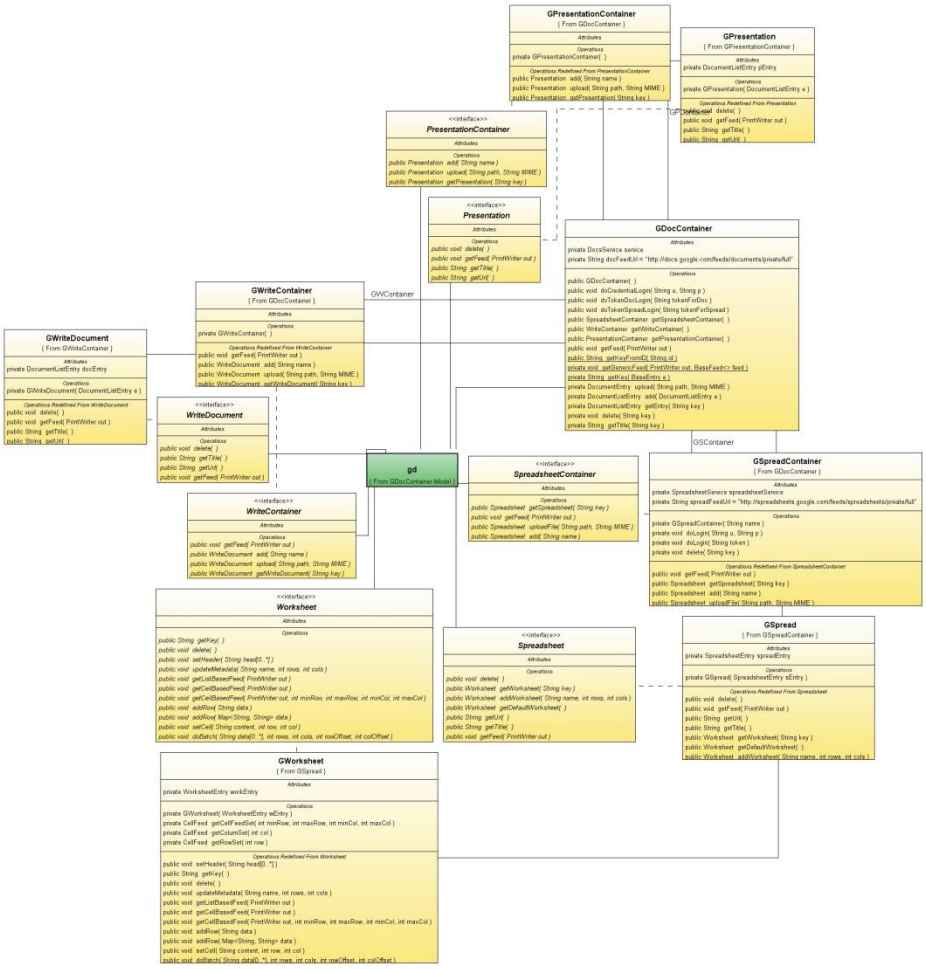


Figura 17

3.1.3 Esempi di funzionamento

Per quanto riguarda il funzionamento, la componente può essere utilizzata in qualsiasi contesto di sviluppo in cui si faccia uso del linguaggio Java, tuttavia gli esempi creati durante questo stage sono prettamente orientati alle applicazioni Web. In particolare tutti i test sono stati effettuati all'interno di *servlet* costruite appositamente per lo scopo, e contenute nel Web container, il cui modello di sviluppo è descritto dal codice seguente.

```
public class Servlet extends SPServlet implements SPInvokable {
  public void doProcess(HttpServletRequest request,
HttpServletResponse response) {

  }
}
```

Molto semplicemente si tratta di una singola classe con un unico metodo al suo interno, attraverso il quale è possibile controllare le richieste dal browser con l'oggetto *HttpServletRequest* e gestire le risposte tramite l'oggetto *HttpServletResponse*. All'interno di questo metodo ho inserito tutto ciò che riguardava la logica di

un’applicazione, quindi anche le operazioni eseguite tramite la componente. L’uso di questa è relativamente semplice, come è semplice l’idea con la quale è stata disegnata e l’interfaccia offerta per le proprie funzionalità. Tutto il controllo è originato da una istanza della classe *GDocContainer*. La prima operazione da eseguire necessariamente per attivare l’oggetto è l’autenticazione, la quale può essere compiuta alternativamente tramite l’immissione diretta delle credenziali, oppure tramite l’immissione del *token* di sessione ottenuto in seguito all’esecuzione del protocollo *AuthSub* .

```
GDocContainer container=new GDocContainer();
container.doCredentialLogin(“username”,”password”);
```

```
container.doTokenDocLogin(“DocToken”);
container.doTokenSpreadLogin(“SpreadsheetToken”);
```

A questo punto è possibile effettuare qualsiasi operazione, come ad esempio inserire un foglio di calcolo.

```
SpreadContainer sC= container.getSpreadsheetContainer();
Spreadsheet s=sC.add(“Titolo”);
```

Quello che bisogna tenere in considerazione è che il metodo di inserimento è associato ad una particolare istanza di *SpreadContainer*, cioè il contenitore interno. Restituisce un oggetto Spreadsheet associato al documento remoto appena creato, così da poter effettuare le operazioni. Allo stesso modo è possibile ottenere un documento tramite il metodo *getSpreadsheet* al quale viene fornita la chiave relativa come visto nel capitolo 2.2. L’eliminazione invece non è eseguita attraverso il contenitore, ma sull’oggetto stesso come nel seguente esempio.

```
s.delete();
```

È plausibile chiedersi cosa succeda nel caso sia chiamato più volte questo metodo sullo stesso oggetto, cioè in che modo è associata una funzionalità rispettivamente al servizio. Ebbene, dal livello API Google è sollevata una eccezione a significare che il documento è già stato eliminato, quindi l’operazione non può essere portata a termine. Gestendo opportunamente questa eccezione ho reso il metodo idem potente e semplicemente non provoca alcun effetto.
Per quanto riguarda un foglio di calcolo è possibile eseguire molte altre operazioni rispetto ad altri tipi di documento poiché per natura

ha una struttura logica più complessa. Una dimostrazione è la seguente.

```
Worksheet w=s.addWorksheet(“fattura”,5,2);
List<String> header=new LinkedList<String>();
header.add(“articolo”); header.add(“prezzo”);
w.setHeader(header);
w.addRow(“articolo=art1”,”prezzo=10”);
w.addRow(“articolo=art2”,”prezzo=15”);
w.addRow(“articolo=art3”,”prezzo=20”);
w.setCell(“=SUM(B2:B4)”,5,2);
```

L’esempio inizia con la creazione di un nuovo foglio di lavoro all’interno del documento generato precedente. Successivamente viene definita l’intestazione per le colonne alla quale bisogna fare riferimento nel caso l’inserimento avvenga per righe. Infine il metodo *setCell* permette di definire un contenuto generico per la cella in questo caso una funzione di somma che calcola il prezzo totale degli articoli inseriti. È importante notare come la generazione di un documento possa avvenire in maniera completamente dinamica semplicemente parametrizzando alcuni valori, ad esempio la dimensione del foglio di lavoro, la posizione dei dati, e naturalmente i dati inseriti. Un’altra modalità di inserimento è costituita dal metodo *doBatch* il quale si distingue dai precedenti in quanto permette una migliore gestione della comunicazione con il servizio e quindi una maggiore efficienza. In particolare ogni inserimento per riga o per cella costituisce una connessione e tipicamente è necessario fare questo per parecchie volte all’interno di uno stesso documento. L’approccio *batch* invece permette un inserimento multiplo di dati utilizzando una connessione soltanto.

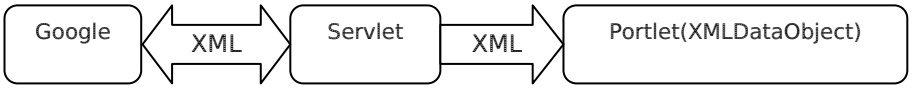
```
List<String> data=new LinkedList<String>();
data.add(“val1”); data.add(“val2”);
data.add(“val3”); data.add(“val4”);
w.doBatch(data,2,2,0,0);
```

Anche questo metodo permette di definire la disposizione dei dati specificando la dimensione della regione sulla quale fare l’inserimento e lo scostamento di questa dalla cella iniziale.

3.1.4 Esempi d’integrazione

Dopo questi semplici utilizzi sono passato a implementare degli esempi pratici di applicazioni Web in grado di utilizzare questa componente. Ho effettuato molti esempi funzionanti, talvolta anche

ripetitivi per verificare le funzionalità all'interno dell'ambiente operativo. Riporto quindi soltanto i più significativi.
Le componenti in generale che ho sviluppato per ogni esempio sono essenzialmente due: le *Portlet* per quanto riguarda la visualizzazione o comunque la logica applicativa lato client, mentre le *servlet* per quanto riguarda il server. Per le prime, come visto nel capitolo 1.2.4, mi sono servito dello strumento *PortalStudio* mentre per le seconde generalmente ho preferito uno sviluppo manuale. Il modello applicativo utilizzato è schematizzato dalla seguente immagine.



In generale la *servlet* si occupa di interagire con il Google provider tramite la componente da me sviluppata e i dati ottenuti vengono poi inoltrati alla *Portlet* per l'elaborazione e la presentazione.
Un primo semplice esempio riguarda proprio la creazione di una *Portlet* in grado di visualizzare la lista dei documenti presenti su un account. Le componenti utilizzate all'interno sono:

- Una griglia collegata ad un *XMLDataObject* per la visualizzazione tabellare del titolo e dalla data di creazione di un documento in cui ogni riga costituisce un link ad esso.
- Un *XMLDataObject* per la lettura dei dati dalla *servlet* e l'estrazione dei campi tramite un comando *XPath*.

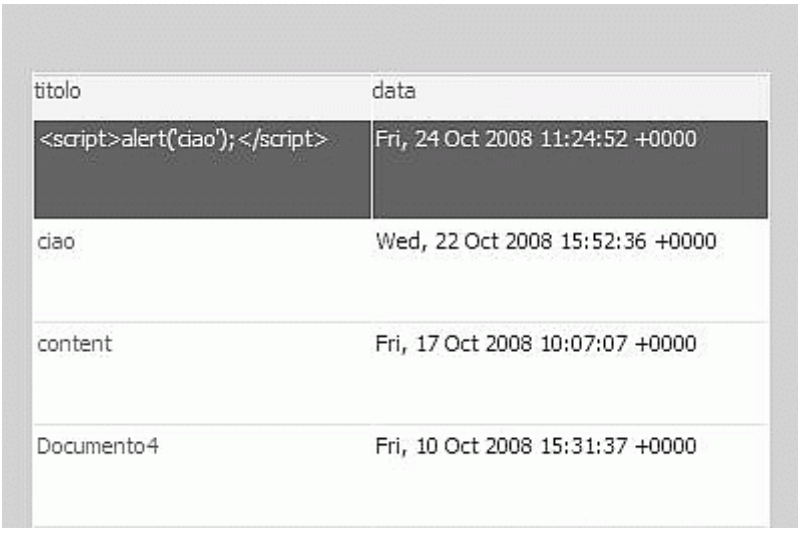
La *servlet*, sviluppata secondo quanto visto negli esempi d'uso della componente Java, molto semplicemente ottiene un feed dal provider Google e costituisce la sorgente per l'oggetto XMLDataObject.

```
public class Proxy extends SPServlet implements SPInvokable {
  public void doProcess(HttpServletRequest request,
HttpServletResponse response) {
    PrintWriter out=response.getWriter();
    GDocContainer container=new GDocContainer();
    container.doCredentialLogin("dcompagn.stage@gmail.com","pswd");
    container.getFeed(out);
  }
}
```

Ottiene ad esempio:

```
<rss xmlns:atom='http://www.w3.org/2005/Atom'
xmlns:openSearch='http://a9.com/-/spec/opensearchrss/1.0/'
xmlns:gd='http://schemas.google.com/g/2005' version='2.0'>
<channel>
<lastBuildDate>Fri, 10 Oct 2008 14:09:21 +0000</lastBuildDate>
<title>Available Documents</title>
<description/>
<link>http://docs.google.com</link>
<managingEditor>dcompagn.stage@gmail.com</managingEditor>
<openSearch:totalResults>3</openSearch:totalResults>
<item>
<guid isPermaLink='false'>
http://docs.google.com/feeds/documents/private/full/document%3Add79pp22_52mj
xsmc9 </guid>
<pubDate>Fri, 24 Oct 2008 11:24:52 +0000</pubDate>
<atom:updated>2008-10-09T10:31:52.733Z</atom:updated>
<category domain='http://schemas.google.com/g/2005#kind'>
http://schemas.google.com/docs/2007#document</category>
<title><scipt>alert('ciao');</script></title>
<enclosure type='text/html'
url='http://docs.google.com/feeds/download/documents/RawDocContents?action=f
etch&amp;justBody=false&amp;revision=_latest&amp;editMode=false&amp;docID=dd
79pp22_52mjxsmc9' length='0'/>
<link>http://docs.google.com/Doc?id=dd79pp22_52mjxsmc9</link>
<author>dcompagn.stage@gmail.com (dcompagn.stage)</author>
<gd:feedLink rel='http://schemas.google.com/acl/2007#accessControlList'
href='http://docs.google.com/feeds/acl/private/full/document%3Add79pp22_52mj
xsmc9'/>
</item>
....ecc...
</channel>
</rss>
```

Dal quale vengono estratti i campi *title* e *pubDate* tramite *l'XMLDataObject* e visualizzato il risultato nella griglia di Figura 18.



titolo	data
<script>alert('ciao');</script>	Fri, 24 Oct 2008 11:24:52 +0000
ciao	Wed, 22 Oct 2008 15:52:36 +0000
content	Fri, 17 Oct 2008 10:07:07 +0000
Documento4	Fri, 10 Oct 2008 15:31:37 +0000

Figura 18

Un altro esempio molto simile è costituito da una *Portlet* principale in cui vi sono situati dei tasti come in Figura 19. Ognuno dei quali definisce un'operazione.

Figura 19

Alla pressione del tasto viene eseguita l'azione corrispondente tramite una *servlet* simile a quella dell'esempio precedente. L'associazione tra il tasto e l'azione è effettuata su client tramite del codice Javascript molto semplice che attribuisce inoltre un valore ad una variabile *action*, anch'essa contenuta nella *Portlet*, corrispondente all'operazione richiesta. L'indirizzo con cui viene richiamata la *servlet* diventa quindi parametrizzata su questa variabile. Per esempio:

```
http://localhost/startServlet?action=”azione da eseguire”
```

Il resto avviene in maniera molto simile a quanto visto prima.

Un esempio leggermente diverso è stato sviluppato per testare l'integrazione con lo strumento di Zoom descritto nel capitolo 1.2.3. Questo strumento permette una gestione avanzata del contenuto della base di dati attraverso una potente interfaccia dinamica con la quale è possibile visualizzare, aggiungere, rimuovere e filtrare informazioni secondo la definizione di alcuni parametri. In Figura 20 è mostrato un esempio di Zoom. In questo esempio ho dapprima proceduto allo sviluppo di una applicazione ex novo, tramite lo strumento *SitePainter*, nella quale ho definito:

- La struttura della base di dati
- Una routine per l'inserimento dei dati nell'applicazione
- Una *servlet* per l'esecuzione delle funzionalità
- Una *Portlet* per la creazione di nuovi documenti e l'esecuzione della routine

La base di dati molto semplicemente contiene una tabella per la memorizzazione dei documenti attraverso i campi: titolo,data di creazione, creatore e URL. Quest'ultimo è stato poi utilizzato come collegamento al documento dall'interno dello Zoom. La *Portlet* rappresenta il collante tra la *servlet* incaricata di creare il documento e la routine incaricata di inserire i campi relativi nella base di dati ed è composta dalle seguenti parti:

- Una casella di testo per l'immissione del titolo
- Tre tasti associati all'inserimento dei diversi tipi di documento
- Due variabili per il controllo della *servlet*
- Quattro variabili per il controllo della routine
- Un *XMLDataObject* per l'esecuzione della *servlet*
- Un *SPLinker* per l'esecuzione della routine

Il funzionamento della *servlet* è pressoché identico a quello degli esempi precedenti, in cui vi è l'utilizzo della componente, se non per il fatto che riceve due parametri in ingresso associati alle variabili della *portlet*: un parametro *action* che associa la creazione al tipo corretto di documento e un parametro *title* che assegna il titolo.

```
http://localhost/insertServlet?action="ins_doc"&title="titolo"
```

Restituisce alla *portlet* un *feed* contenente i dettagli relativi alla creazione avvenuta.

La routine definisce una *query* per l'inserimento nella base di dati parametrizzata su quattro valori rispettivi alle variabili contenute nella *portlet*.

La sequenza delle operazioni diventa quindi la seguente:

1. Alla pressione di uno dei tasti è assegnato a una variabile "v_action" il comando da eseguire.
2. Viene poi richiamata tramite l'oggetto XMLDataObject una servlet cui sono passati la variabile definita prima e il contenuto della casella di testo come secondo parametro.
3. L'XMLDataObject ottiene un *feed* dalla *servlet* contenente le informazioni relative al documento creato.
4. Da questo vengono estratti nelle rispettive quattro variabili il titolo, il creatore, la data, e l'url tramite un semplice comando *XPath*.
5. Viene richiamata la routine per mezzo dell'SPLinker al quale vengono attribuite le variabili prima definite.
6. Lo Zoom viene aggiornato per visualizzare correttamente i dettagli relativi all'inserimento effettuato come in Figura 20.

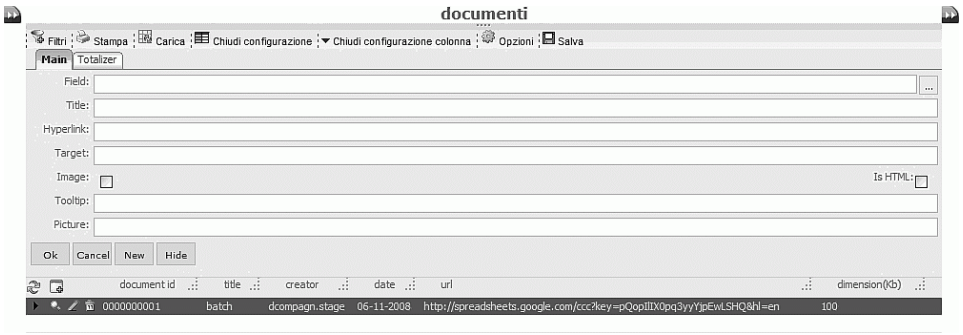


Figura 20

All'interno dello Zoom è stato poi definito tramite l'URL un collegamento al documento che se cliccato richiama l'apertura di questo in una nuova finestra del browser.

Come si è potuto vedere dagli esempi presentati il modello di funzionamento di un'applicazione è sempre il medesimo. In generale la *Portlet* predispone, attraverso gli oggetti al suo interno, l'interfaccia utente per controllare le operazioni della componente che è collocata nel server.

3.1.5 Integrazione in SitePainter

L'esempio appena spiegato dimostra come sia possibile in modo sufficientemente completo l'integrazione tra un applicativo e *Google Document* attraverso la componente. Ciò che rimane ancora insoddisfacente è l'utilizzo di questa all'interno della servlet, che necessariamente deve essere creata a mano. Per questo motivo è iniziato lo sviluppo di una libreria attraverso lo strumento *LibraryEditor* che consente di utilizzare le medesime funzionalità della componente all'interno di *SitePainter*, in particolare durante la creazione di routine. Come visto nel capitolo 2.1.1 la generazione di una routine corrisponde a tutti gli effetti alla creazione di una *servlet* è quindi è possibile sfruttare questo meccanismo per implementare qualsiasi tipo di applicazione come gli esempi precedenti. Naturalmente una *servlet* generata a partire da una routine indirettamente contiene il codice della libreria quindi quello della componente.

Ne analizzo ora la struttura secondo quanto definito nel capitolo 2.3.

```
public class GoogleStatic {
  static GDocContainer Container=new GDocContainer();
  // Static Methods
}
```

La libreria rappresenta una classe al cui interno è presente soltanto un oggetto della classe *GDocContainer*. Quindi è costruita attorno alla componente in modo da utilizzare il più possibile il codice già scritto. Lo scheletro è generato automaticamente dal *LibraryEditor* una volta che sono stati inseriti tutti i prototipi dei metodi tramite un'apposita interfaccia mostrata in Figura 21.

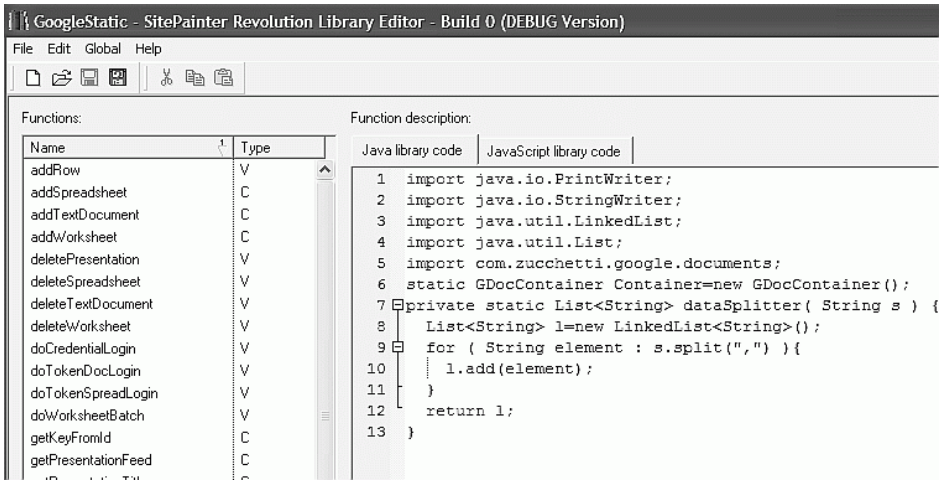


Figura 21

Ogni prototipo deve specificare il tipo dei parametri e il tipo del valore ritornato in modo da poter essere verificato all'interno di *SitePainter*. I tipi possibili oltre che void sono string, integer, float boolean, ovvero tipi semplici. Ogni metodo deve rispettare questi vincoli per questo è stato necessario adattare l'intera libreria. La struttura a contenitori tipica della componente è stata nascosta e sostituita dall'utilizzo di metodi statici. Inoltre ogni PrintWriter è stato convertito in stringa tramite oggetti StringWriter secondo l'esempio seguente.

```
//---Start function: getSpreadsheetFeed
public static String getSpreadsheetFeed(String key) {
  try {
    StringWriter sW=new StringWriter();
    PrintWriter pW=new PrintWriter(sW);
    Container.getSpreadsheetContainer().getSpreadsheet(key)
    .getFeed(pW);
    return sW.toString();
  } catch ( Exception e ) {}
  return "";
}
//---End function
```

Mentre per quanto riguarda i parametri di tipo lista la generazione avviene a partire da stringhe semplici tramite una semplice funzione di *splitting* che utilizza il carattere virgola come separatore.

```
private static List<String> dataSplitter( String s ) {
  List<String> l=new LinkedList<String>();
  for ( String element : s.split(",") ){
    l.add(element);
  }
  return l;
}
```

La lista dei metodi pubblici che è possibile utilizzare diventa la seguente:

ID	Nome	Descrizione
1	doCredentialLogin	effettua l'autenticazione all'account google tramite username e password
2	doTokenDocLogin	effettua l'autenticazione al servizio documenti tramite token
3	doTokenSpreadLogin	effettua l'autenticazione tramite token per il servizio relativo agli spreadsheet
4	getSpreadsheetsFeed	ottiene il feed RSS degli spreadsheet contenuti nell'account
5	uploadSpreadsheetFile	upload un file tipo spreadsheet
6	addSpreadsheet	aggiunge uno spreadsheet
7	deleteSpreadsheet	elimina uno spreadsheet
8	addWorksheet	aggiunge un worksheet all'interno dello spreadsheet specificato e ne ritorna la chiave identificativa
9	getSpreadsheetUrl	ottiene l'URL dello spreadsheet identificato
10	getSpreadsheetTitle	ottiene il titolo dello spreadsheet identificato
11	getSpreadsheetFeed	ottiene il feed relativo al singolo spreadsheet
12	deleteWorksheet	elimina un worksheet da uno spreadsheet
13	setWorksheetHeader	imposta le celle di header di un worksheet per l'inserimento a righe
14	updateWorksheetMetadata	aggiorna i metadati di un worksheet
15	getWorksheetListBasedFeed	ottiene il feed RSS del worksheet visto per righe
16	getWorksheetCellBasedFeed	ottiene il feed RSS del vorksheet visto per celle
17	getWorksheetRegionCellBasedFeed	ottiene il feed RSS di una regione di worksheet
18	addRow	aggiunge una riga ad un worksheet
19	setCell	inserisce il contenuto nella cella specificata
20	doWorksheetBatch	esegue un inserimento multiplo di

		valori nella regione di worksheet specificata
21	getTextDocumentsFeed	ottiene il feed di un documento di testo
22	addTextDocument	aggiunge un documento di testo
23	uploadTextDocument	fa l'upload di un documento di teso e ritorna l'URL
24	deleteTextDocument	elimina il documento di testo indicato
25	getTextDocumentTitle	ottiene il titolo del documento di testo specificato
26	getTextDocumentUrl	ottiene l'URL del documento specificato
27	getTextDocumentFeed	ottiene il feed RSS relativo al documento di testo specificato
28	uploadPresentation	fa l'upload di una presentazione e ne ritorna l'URL
29	deletePresentation	elimina la presentazione indicata
30	getPresentationFeed	ottiene il feed RSS relativo alla presentazione indicata
31	getPresentationTitle	ottiene il titolo della presentazione
32	getPresentationUrl	ritorna l'URL della presentazione indicata
33	getKeyFromId	ottiene la chiave di un documento dato l'ID contenuto nel tag <id> del feed oppure dall'URL

A questo punto è facile mostrare come l'esempio precedente può essere sviluppato richiamando opportunamente le funzioni dall'interno del *RoutinePainter*. Le variabili contenute nella routine per rendere parametrica la *query* ora non assumono più il valore dei parametri passati dall'SPLinker tramite la richiesta, ma bensì dal risultato dei metodi chiamati come in Figura 22.

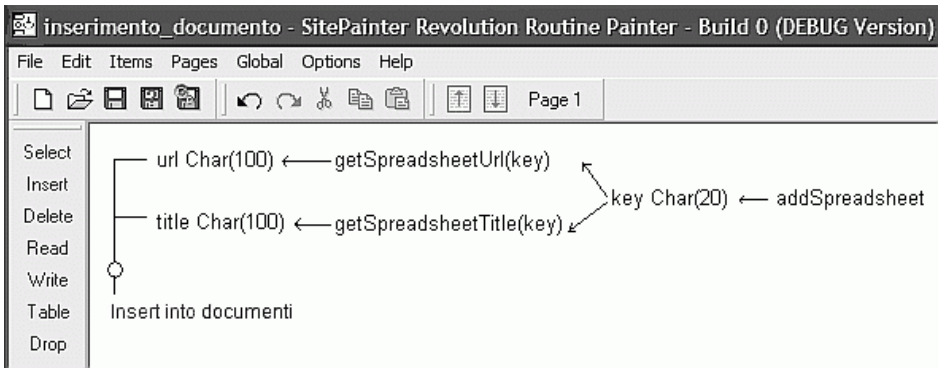


Figura 22

Alla generazione della routine verrà creata una *servlet* al cui interno vi sarà il codice necessario per eseguire tutte le azioni dando pur

sempre la possibilità di ricevere dei parametri attraverso l'oggetto SPLinker in caso di necessità.

3.1.6 Tracciamento dei requisiti

La corrispondenza tra i requisiti definiti al capitolo 2.3 e le funzionalità implementate risulta quindi nella tabella sottostante.

Requisito	Funzione
1.1.1f	4,11,21,27,30
1.1.2f	9,26,32
1.1.3f	5,6,8,22,23,28,33
1.1.4f	10,25,31
1.2.1f	5,6
1.3.1f	22,23
1.4.1f	28
1.2.2f	7
1.3.2f	24
1.4.2f	29
1.2.3f	8
1.2.4f	13,18,19,20
1.2.5f	14
1.2.6f	15,16,17

3.2 Fase II

3.2.1 Considerazioni sulla sicurezza

Alla luce di quanto visto durante lo studio delle problematiche riguardanti la sicurezza è stato analizzato l'impatto che ha l'uso della componente all'interno delle applicazioni Web. Il problema fondamentale di queste come già annunciato deriva dalla necessità di controllare i dati passanti per l'applicazione stessa. Quando si ha a che fare con un sistema che riceve dati da un altro sistema è indispensabile controllare che questi non contengano codice malevolo causando una vulnerabilità di tipo XSS.

Il Cross-site scripting (XSS) è il capostipite degli attacchi ad iniezione di codice in quanto deriva da una scarsa validazione dell'input e può essere combinato con altre vulnerabilità per portare ad una compromissione totale dell'applicazione. Un attacco XSS sfrutta il

concetto di codice nomadico visto in precedenza e si ha quando un'applicazione ottiene forzatamente dei dati contenenti in qualche forma un codice di *scripting* malevolo. Tipicamente si può classificare in non-persistente o Reflected quando i dati in ingresso sono usati immediatamente dal server per generare una pagina risultante, viceversa persistente o *Stored* quando sono memorizzati sul server e solo in un secondo momento visualizzati agli utenti.
Per porre rimedio a questo problema ogni applicativo sviluppato tramite *SitePainter* è dotato di un sanitizzatore dei dati che agisce nel livello di presentazione, ovvero prima che questi siano interpretati dal Web browser. Grazie a questa tecnica non sono stato interessato direttamente dal problema in quanto la componente sviluppata agisce completamente nel server e quindi qualsiasi informazione entrante nell'applicazione è sicuramente filtrata prima di essere visualizzata. A titolo d'esempio è possibile vedere la Figura 18 in cui intenzionalmente ho creato un documento dal titolo:

```
<script>alert('ciao');</script>
```

Questo rappresenta a tutti gli effetti un attacco XSS persistente. La visualizzazione del *feed* contenente la lista dei documenti nella griglia infatti dovrebbe provocare l'apertura di un *alert* Javascript con la stringa relativa se non fosse opportunamente filtrato. Un risultato diverso sarebbe stato ottenuto nel caso la componente fosse stata sviluppata lato client quindi tramite Javascript. I dati sarebbero giunti immediatamente all'applicazione evitando il controllo del sanitizzatore, e quindi interpretati dal browser.

3.2.2 Infrastruttura sperimentale per l'autenticazione

Un'altra problematica analizzata riguarda direttamente il protocollo d'autenticazione visto nel capitolo precedente e la stretta relazione che possiede con la gestione delle sessioni. Il principio su cui si basa l'uso del *token* implica la necessità di un meccanismo che permette di associarlo ad un utente per l'intera durata della sessione secondo quanto visto nel capitolo 2.4.1. Ho creato quindi un'infrastruttura essenziale, costituita da alcune *servlet* in grado di dimostrare il funzionamento di *AuthSub*. Questa rappresenta soltanto un prototipo dal quale partire nel caso l'azienda intenda adottare questa soluzione.
Le parti sviluppate sono essenzialmente due: una *Login servlet* ed una *HandleToken servlet*. Rispettivamente la prima ha il compito di

stabilire la creazione della sessione utente, mentre la seconda ha il compito di associare questa sessione al *token* ottenuto tramite il *Google AuthSub Provider*. Prima di descrivere la sequenza delle operazioni è necessario spiegare perché vi sia l'utilizzo di due *token* distinti. Uno rappresenta quello ottenuto tramite *AuthSub* e viene chiamato *AuthSubToken* mentre il secondo è un *token* generato dall'applicazione per associare l'utente al proprio *AuthSubToken*. Ecco nel dettaglio cosa avviene:

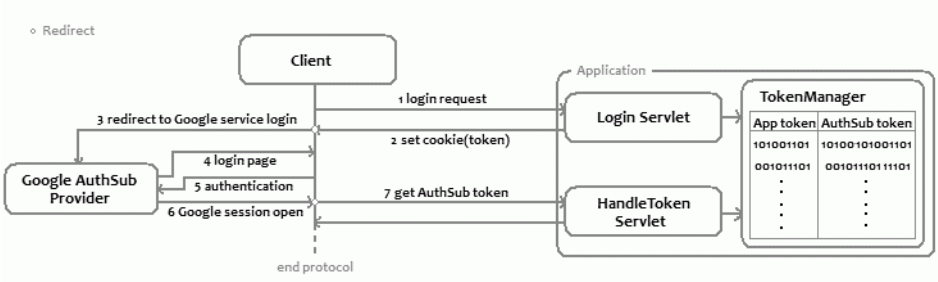


Figura 23

Inizialmente l'utente richiede l'accesso all'applicazione, supponendo che questo non sia già autenticato viene creata dalla *LoginServlet* una nuova sessione generando un *token* casuale che andrà a costituire il contenuto del cookie nel browser dell'utente. Allo stesso tempo genera anche un URL sul quale reindirizzare l'utente per effettuare il login. Una volta che il browser crea fisicamente il cookie, viene ridiretto verso la pagina di login corrispondente ai punti 1,2 e 3 dell'analisi del protocollo *AuthSub*. A questo punto quando l'utente conferma l'autenticazione immettendo le proprie credenziali, riceve *l'AuthSubToken* e viene ridiretto alla *servlet HanleToken*. La quale dapprima chiede la trasformazione di questo a *token* di sessione, come visto in precedenza, e successivamente lo associa al *token* generato inizialmente dalla *LoginServlet*, tramite una struttura dati di supporto implementata da una classe *TokenManager*. Alla fine di questa sequenza l'applicazione è in grado di autenticare tutte le richieste verso il Google Provider e quindi di accedere ai servizi.

Il diagramma di sequenza seguente schematizza la sequenza delle operazioni. La *RetriveFeedServlet* si tratta semplicemente di un *servlet* d'esempio in grado di comunicare con il *GoogleProvider* una volta che l'autenticazione è stata compiuta.

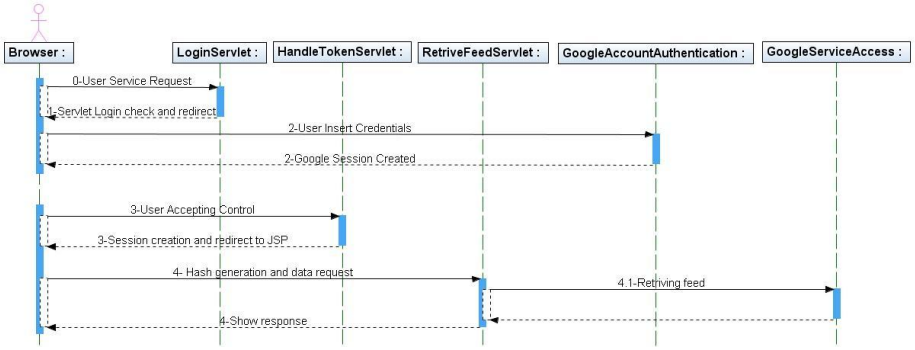


Figura 24

Un compito molto importante è garantire che nessuno dei due token possa essere recuperato da altri utenti, poiché avrebbero immediatamente accesso sia all'applicazione che al servizio Google.

Per quanto riguarda *l'AuthSubToken* che è memorizzato esclusivamente nel server è sufficiente adottare l'utilizzo di un protocollo cifrato in maniera da evitare che qualcuno possa intercettarlo durante lo scambio. Per il *token* contenuto nel browser invece il problema è più serio. È necessario innanzi tutto garantire che l'applicazione non sia vulnerabile ad attacchi XSS in quanto tramite l'esecuzione di codice Javascript è possibile ottenere il contenuto di un cookie. Inoltre è indispensabile impedire che un altro sito possa sfruttare la sessione aperta nel browser per effettuare richieste indirettamente. Questa vulnerabilità prende il nome di XSRF (Cross-site request forgery) che letteralmente sta per falsificazione tramite richiesta incrociata. È un tipo di attacco in cui un sito Web sfrutta la sessione di un altro sito Web per compiere delle azioni. Contrariamente al cross-site scripting, che sfrutta la fiducia di un utente rispetto ad un particolare sito, questo sfrutta la fiducia di un sito verso un particolare utente e accade principalmente per due motivi: uno rappresentato dal fatto che il browser è un ambiente aperto e permette la navigazione a più finestre non garantendo una adeguata protezione all'interferenza che queste possono avere nella gestione di dati condivisi. Il secondo motivo invece è che la *Same Origin policy* non vieta ad un sito di contenere oggetti provenienti da altri siti. Supponendo uno scenario in cui esistano due domini "bob.com" e "trudy.com" e supponendo che il primo offra un URL del tipo:

"http://bob.com/aggiungi?email=alice@gmail.com"

che permetta di aggiungere l'indirizzo mail di Alice all'agenda. Questo potrebbe costituire una semplice *servlet* come quelle

sviluppate nei miei esempi. Un attacco potrebbe figurare come segue:

- Alice si autentica in “bob.com” e ottiene l’apertura di una sessione
- In seguito entra in “trudy.com”, avendo comunque la sessione precedente aperta, e in quest’ultimo è presente un elemento HTML con un attributo “src” (ad es. un’immagine invisibile) che acceda a:
  “http://bob.com/aggiungi?email=spam@spam.com”
- Il browser quindi carica l’indirizzo autenticato dalla sessione ancora aperta.

Il risultato quindi sarà che il sito di Bob è stato modificato da Trudy sfruttando la sessione di Alice.

Per prevenire ciò è necessario che ogni richiesta effettuata verso l’applicazione possieda oltre al cookie relativo alla sessione un’informazione aggiuntiva calcolata dinamicamente. Una buona tecnica è associare un *hash* del contenuto del cookie ad un *time-stamp* anche per garantire la freschezza della richiesta ed evitare *replay-attack*. Poiché l’attaccante non è in grado di processare queste informazioni ogni richiesta sarà scartata. Questo protocollo implica che ogni pagina client contenga il codice necessario a generare queste informazioni ed inviarle al server ad esempio come parametri di una richiesta GET e naturalmente il server deve contenere le istruzioni per confrontare ogni richiesta ricevuta secondo le informazioni in suo possesso. In particolare agli esempi svolti durante questo stage ho apportato delle modifiche alle *Portlet* sviluppate con *PortalStudio* predisponendo all’interno il seguente codice:

```
<%
String cookie= request.getCookies().getValue();
String timestamp= new Date()).getTime().toString();
String dati=cookie+timeStamp;
MessageDigest digest=MessageDigest.getInstance("MD5");
String hash=digest.digest(dati.getBytes()).toString();
%>
var timestamp=<%=timeStamp%>;
var hash=<%=hash%>;
```

Ogni *Portlet* come già visto costituisce una pagina JSP, quindi le linee di codice incluse tra i marcatori <% e %> rappresentano istruzioni Java e sono elaborate dal server. Ciò che viene calcolato è proprio un *hash* tra il contenuto del cookie ed il *time-stamp*. Quest’ultimo naturalmente deve essere inviato anche in chiaro per verificare che non vi sia stata contraffazione della richiesta. L’algoritmo utilizzato

per calcolare ciò è semplicemente MD5 ed è fornito di base dal linguaggio Java. I due valori generati sono poi inseriti in altrettante variabili Javascript che sono utilizzate dalle varie componenti di una *Portlet* per compiere richieste autenticate.
Allo stesso modo ogni *servlet* deve includere il seguente codice per eseguire i controlli necessari:

```
long ts = Long.parseLong(timeStamp);
long currentTime = new Date().getTime();
if ((currentTime - createTime) > DURATA ) { throw new
SecurityException(); }
String cookie= request.getCookies().getValue();
String dati = cookie+ timeStamp;
String dati=cookie+timeStamp;
MessageDigest digest=MessageDigest.getInstance("MD5");
String nuovoHash=digest.digest(dati.getBytes()).toString();
If(hash.equals(nuovoHash)==false) throw new SecurityException();
```

Si suppone che la variabile *timeStamp* e *hash* siano i parametri ottenuti dalla richiesta della *Portlet*. Viene controllato dapprima che la richiesta sia stata effettuata entro una finestra di tempo definita dalla costante DURATA e in caso positivo viene ricalcolato un nuovo *hash* dai parametri ricevuti. I due valori devono essere perfettamente identici altrimenti significa che vi è stato un problema.

4 Conclusioni

4.1 Considerazioni finali

Sebbene da parte mia il lavoro svolto sia stato portato a termine con il massimo impegno ed un approccio critico, purtroppo non posso concludere di aver creato un prodotto facilmente ed immediatamente utilizzabile dall'azienda. Questo non è stato determinato dalla mancanza di conoscenze o tecnologie e neppure dalla scarsa assistenza del tutor aziendale ma dai limiti che il sistema oggetto di questo stage, ovvero Google, pone ai propri servizi. L'impressione avuta sia dal sottoscritto che dal responsabile aziendale è che Google si affacci agli sviluppatori imponendo fortemente le modalità d'uso dei propri servizi, lasciando talvolta poca libertà di implementazione. Nel mio caso, l'uso di *AuthSub* ha limitato molto lo sviluppo di una componente completa ed autonoma poiché il funzionamento è vincolato,come già visto, dall'infrastruttura necessaria a gestire il protocollo di autenticazione. Un altro aspetto assolutamente da considerare è la dipendenza della componente sviluppata verso le API di Google. È stato notato come l'aggiornamento a nuove versioni sia abbastanza frequente: soltanto durante il periodo di stage ciò si è verificato due volte. Questo potrebbe essere fonte di problemi nel caso Google decida di non rendere retro compatibili le varie versioni. Ad ogni modo le versioni usate in questo progetto sono la 1.22 e la 1.24.

Le difficoltà riscontrate hanno coinvolto le fasi iniziali del progetto e sono state causate principalmente dalla quasi totale assenza di documentazione in merito alle librerie del sistema *Google Document*. In molte occasioni ciò mi ha costretto ad addentrarmi nell'analisi del codice sorgente di quest'ultime, costringendo all'esecuzione parallela di più attività come sarà mostrato nel piano di lavoro, causando un rallentamento dello svolgimento dell'intero progetto ma che non ha pregiudicato il raggiungimento degli obbiettivi. C'è da sottolineare come la componente sviluppata soddisfi appieno i requisiti delineati durante le prime fasi del progetto e quindi risulta funzionante in tutte le sue parti. Rimane soltanto il rammarico di non essere riuscito ad integrare completamente il funzionamento nel sistema aziendale a causa, appunto, del sistema di autenticazione. In ogni caso il materiale prodotto, cioè la componente, gli esempi e la relativa

documentazione è stato consegnato all'azienda, assieme inoltre ad un documento in cui vengono riassunte le principali vulnerabilità di sicurezza a cui sono soggette le applicazioni Web.
Nonostante le problematiche riscontrate ed evidenziate, l'attività di stage è stata sicuramente positiva e utile sia per vedere applicate ad un contesto concreto e reale le conoscenze acquisite in ambito universitario sia per acquisirne di nuove. A tutto ciò ha sicuramente contribuito il clima di cordialità e disponibilità che ho incontrato nel periodo trascorso all'interno dell'azienda.

4.2 Conoscenze acquisite e utilizzate

Questo stage è stata un'ottima occasione per venire a conoscenza delle idee e delle tecniche più recenti utilizzate dall'azienda Zucchetti S.p.A nello sviluppo automatizzato di applicazioni Web gestionali complesse. Le tematiche coinvolte in questa esperienza riguardano principalmente la programmazione orientata al Web attraverso la piattaforma *Java Enterprise*. Al fine di analizzare al meglio l'ambiente operativo, ho dovuto procedere allo studio di tecnologie quali JSP, per la creazione di contenuti HTML dinamici, e *servlet*, per implementare la logica delle applicazioni. Chiaramente questo ha implicato anche lo studio dei principi che governano queste tecnologie e gli strumenti che ne permettono l'utilizzo, come il Web container. Ogni elemento applicativo sviluppato come esempio fa proprio uso di queste tecnologie. Ho potuto apprendere anche l'utilizzo del linguaggio di *scripting* Javascript per le operazioni eseguite client-side, ovvero nel browser, soltanto in forma limitata. Nonostante ciò, questo linguaggio è risultato molto semplice da capire e mi ha permesso di controllare gli eventi associati alle componenti di una *Portlet*. Indirettamente, ho potuto anche osservare il comportamento dell'architettura AJAX all'interno delle applicazioni Web. Un altro argomento che ho avuto modo di approfondire è stato il controllo del protocollo HTTP attraverso il linguaggio Java. La formazione ricevuta con i corsi universitari su questo argomento è stata per lo più di carattere teorico, qui invece mi son ritrovato a toccare con mano il problema.
Infine uno sguardo importante è stato rivolto al mondo dei servizi Google, le infrastrutture, le tecnologie e i principi che rendono questa società così importante nel Web.
Le conoscenze acquisite durante questo corso di laurea sono state adeguate in ogni occasione e mi hanno permesso di esprimermi al meglio durante tutto lo stage, in particolare ho avuto modo di

applicare conoscenze relative ai corsi di programmazione 3 per la scrittura del codice, basi di dati 2, per l'utilizzo del linguaggio XML, e sicurezza, per lo studio approfondito del protocollo di autenticazione *AuthSub*. Ritengo che la formazione universitaria in ogni caso sia stata indispensabile sia per affrontare criticamente il problema sia per organizzare al meglio le attività svolte.
La tabella seguente cerca di riassumere tutti i corsi che sono stati di utilità, anche marginale, al fine dello svolgimento dello stage.

Corso	Utilità
Programmazione 3	Fondamentale per lo sviluppo della componente, per la creazione delle Servlet e delle pagine JSP.
Ingegneria del Software	Per un approccio metodico allo sviluppo e la definizione delle attività, nonché la stesura della documentazione.
Sicurezza	Per l'analisi del protocollo di autenticazione AuthSub e l'implementazione dell'infrastruttura in grado di supportarlo.
Tecnologie Web, Basi di Dati 2, Reti	Per quanto riguarda l'analisi del servizio Google Document, il protocollo, e le API che ne permettono l'accesso
Inglese	Per la comprensione della documentazione utilizzata

4.3 Pianificazione settimanale

Il tempo necessario stimato per l'esecuzione di questo stage è stato pari a 300 ore, 250 delle quali impiegate nello sviluppo e suddivise come di seguito, mentre le rimanenti 50 utilizzate per la stesura della tesi.
L'orario di lavoro (9:00-13:00,14:00-18:00) ha permesso di effettuare un quantitativo di 40 ore settimanali in azienda, occupando quindi 6 settimane a partire dal giorno 6 ottobre 2008.
Le attività pianificate sono state le seguenti:

Settimana	
1	➢ Studio di fattibilità e analisi dei requisiti. ➢ Studio delle API offerte da Google per la gestione dei documenti.

	➢ Analisi delle Google API. ➢ Studio delle funzionalità principali per la gestione dei documenti. ➢ Studio dell'ambiente operativo e le origini dei dati. ➢ Studio dell'architettura delle applicazioni Web.
2	➢ Scelta e impostazione dell'ambiente di sviluppo. ➢ Sviluppo di qualche applicativo d'esempio. ➢ Progettazione della componente. ➢ Definizione della specifica tecnica e delle funzionalità secondo i requisiti. ➢ Stesura di un prototipo per la dimostrazione di parte delle funzionalità richieste.
3	➢ Panoramica generale sulla sicurezza nelle applicazioni Web e principali vulnerabilità. ➢ Autenticazione al servizio Google Document.
4	➢ Sviluppo e verifica delle funzionalità della componente.
5	➢ Completamento dello sviluppo. ➢ Revisione, e sviluppo di qualche esempio d'uso.
6	➢ Stesura della documentazione.

Il piano di lavoro è stato rispettato senza stravolgimenti di rilievo. L'unica variazione rilevante è stata la durata dell'apprendimento d'uso delle API Google che ha interessato le prime tre settimane, durante le quali vi è stata un'esecuzione parallela di più attività. Questo ha comportato un piccolo ritardo nello sviluppo della componente che quindi è stato concluso durante l'ultima settimana.
L'immagine sotto rappresenta il diagramma di *Gantt* del lavoro svolto.

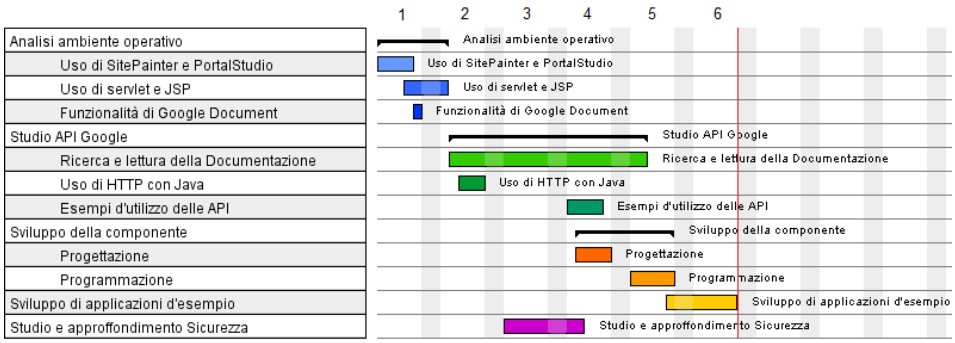


Figura 25

4.4 Glossario

AJAX: Acronimo di Asynchronous JavaScript and XML, è una tecnica di sviluppo web per creare applicazioni web interattive. L'intento di tale tecnica è quello di ottenere pagine web che rispondono in maniera più rapida, grazie allo scambio in background di piccoli pacchetti di dati con il server, cos che l'intera pagina web non debba essere ricaricata ogni volta che l'utente effettua una modifica. Questa tecnica riesce, quindi, a migliorare l'interattività, la velocità e l'usabilità di una pagina web. AJAX è asincrono nel senso che i dati extra sono richiesti al server e caricati in background senza interferire con il comportamento della pagina esistente. Normalmente le funzioni richiamate sono scritte con il linguaggio JavaScript.

HTML: Acronimo di Hyper Text Mark-up Language è un linguaggio per la presentazione di pagine web.

JavaScript: È un linguaggio di scripting orientato agli oggetti comunemente usato nei siti web. La caratteristica principale di JavaScript è quella di essere un linguaggio interpretato. Il codice quindi non viene compilato bensì c'è un interprete (in JavaScript lato client è il browser che si sta utilizzando) che esegue riga per riga in modalità runtime quanto trascritto nello script.

Java Enterprise Edition : È la versione enterprise della piattaforma java. Essa `e costituita da un insieme di specifiche che definiscono le caratteristiche e le interfacce di un insieme di tecnologie pensate per la realizzazione di applicazioni di tipo enterprise e mission critical.

JSP: JavaServer Pages è una tecnologia Java per lo sviluppo di applicazioni Web che forniscono contenuti dinamici in formato HTML o XML. Si basa su un insieme di speciali tag con cui possono essere invocate funzioni predefinite o codice Java. In aggiunta, permette di creare librerie di nuovi tag che estendono l'insieme dei tag standard. Le librerie di tag JSP si possono considerare estensioni indipendenti

dalla piattaforma delle funzionalità di un Web server. Nel contesto della piattaforma Java, la tecnologia JSP è correlata con quella dei servlet. All'atto della prima invocazione, le pagine JSP vengono infatti tradotte automaticamente da un compilatore JSP in servlet. Una pagina JSP può quindi essere vista come una rappresentazione ad alto livello di un servlet.

UML: Unified Modelling Language. Linguaggio di progettazione che tramite l'utilizzo di schemi permette di descrivere l'architettura di un software in modo univoco e dettagliato.

Macchina virtuale : Indica un software che crea un ambiente simulato ove è possibile utilizzare altro software, come un sistema operativo o applicazioni.

Sandbox: Sta ad indicare un ambiente controllato nel quale vengono eseguiti dei programmi.

Query: Richiesta che viene fatta ad una base di dati utilizzando il linguaggio SQL.

Web application: Applicazione web. Indica un'applicazione scritta per eseguire su di un web server in remoto ed essere eseguita tramite un qualunque browser, rendendo di fatto l'applicazione indipendente dalla piattaforma e completamente centralizzata sul server.

4.5 Riferimenti

Zucchetti S.p.A

http://www.zucchetti.it

Google Document

http://docs.google.com

Google Data APIs

http://code.google.com/intl/it-IT/apis/gdata/

http://groups.google.com/group/Google-Docs-Data-APIs

http://code.google.com/intl/it-IT/apis/gdata/javadoc/

Protocollo HTTP

http://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html

http://java.sun.com/j2se/1.4.2/docs/api/java/net/HttpURLConnection.html

Principi dell'architettura REST

http://www.xml.com/pub/a/2004/12/01/restful-web.html

Linguaggio Javascript

http://www.w3schools.com/js/default.asp

Wikipedia

http://it.wikipedia.org

Guida JSP e Servlet
http://java.html.it/guide/lezione/3420/servlet-e-jsp/
http://java.html.it/guide/leggi/23/guida-jsp/
Open Web Application Security Project
http://www.owasp.org
The Web Application Hacker's Handbook
http://portswigger.net/wahh/

4.6 Ringraziamenti

Ringrazio tutto il personale dell'azienda Zucchetti presso la quale ho lavorato per la sua cordialità, in particolare il responsabile Gregorio Piccoli, il relatore prof. Filè Gilberto per avermi seguito durante questo periodo e per avermi dato molti consigli, la mia famiglia, e tutti gli amici per avermi sostenuto. Infine, un ringraziamento veramente particolare a Silvia.